Error Awareness and Recovery in Conversational Spoken Language Interfaces

Dan Bohus

May 2007
CMU-CS-07-124

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA, 15213

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Thesis Committee:
Alexander I. Rudnicky, *Chair*
Roni Rosenfeld, *Co-chair*
Jeff Schneider
Eric Horvitz, Microsoft Research

This research was sponsored by SRI International subcontract nos. 55-000691 and 03-0002111 under contract no. NBCH-D-03-0010 from the Department of Interior and by the Space and Naval Warfare Systems Center, San Diego, under grant no. N66001-99-1-8905. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either express or implied, of any sponsoring institution, the U.S. government, or any other entity.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
MAY 2007		**00-00-2007 to 00-00-2007**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Error Awareness and Recovery in Conversational Spoken Language Interfaces	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Carnegie Mellon University,School of Computer Science,Pittsburgh,PA,15213**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
see report

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**278**	

Keywords: spoken dialog systems, conversational spoken language interfaces, error detection, error recovery strategies, error recovery policies, dialog management, RavenClaw, implicitly-supervised learning

To my parents,

Abstract

One of the most important and persistent problems in the development of conversational spoken language interfaces is their lack of robustness when confronted with understanding-errors. Most of these errors stem from limitations in current speech recognition technology, and, as a result, appear across all domains and interaction types. There are two approaches towards increased robustness: prevent the errors from happening, or recover from them through conversation, by interacting with the users.

In this dissertation we have engaged in a research program centered on the second approach. We argue that three capabilities are needed in order to seamlessly and efficiently recover from errors: (1) systems must be able to **detect the errors**, preferably as soon as they happen, (2) systems must be equipped with a rich repertoire of error **recovery strategies** that can be used to set the conversation back on track, and (3) systems must know how to choose optimally between different recovery strategies at run-time, i.e. they must have good error **recovery policies**. This work makes a number of contributions in each of these areas.

First, to provide a real-world experimental platform this error handling research program, we developed RavenClaw, a plan-based dialog management framework for task-oriented domains. The framework has a modular architecture that decouples the error handling mechanisms from the domain-specific dialog control logic; in the process, it lessens system authoring effort, promotes portability and reusability, and ensures consistency in error handling behaviors both within and across domains. To date, RavenClaw has been used to develop and successfully deploy a number of spoken dialog systems spanning different domains an interaction types. Together with these systems, RavenClaw provides the infrastructure for the error handling work described in this disseratation.

To **detect errors**, spoken language interfaces typically rely on confidence scores. In this work we investigated in depth current supervised learning techniques for building error detection models. In addition, we proposed a novel, implicitly-supervised approach for this task. No developer supervision is required in this case; rather, the system obtains the supervision signal online, from naturally-occurring patterns in the interaction. We believe this learning paradigm represents an important step towards constructing autonomously self-improving systems. Furthermore, we developed a scalable, data-driven approach that allows a system to continuously monitor and update beliefs throughout the conversation; the proposed approach leads to significant improvements in both the overall effectiveness and efficiency of the interaction.

We developed and empirically investigated a large set of **recovery strategies**, targeting two types of understanding-errors that commonly occur in these systems: misunderstandings and non-understandings. Our results add to an existing body of knowledge about the advantages and disadvantages of these strategies, and highlight the importance of good recovery policies.

In the last part of this work, we proposed and evaluated a novel online-learning based approach for developing **recovery policies**. The system constructs runtime estimates for the likelihood of success of each recovery strategy, together with confidence bounds for those estimates. These estimates are then used to construct a policy online, while balancing the system's exploration and exploitation goals. Experiments with a deployed spoken dialog system showed that the system was able to learn a more effective recovery policy in a relatively short time period.

Acknowledgements

I would like to thank my advisors, Alex Rudnicky and Roni Rosenfeld for their invaluable help, guidance and support. Without them, this body of work would have not been possible. My two other thesis committee members, Jeff Schneider and Eric Horvitz have also provided very valuable insights and advice that have shaped this dissertation. This work also owes a lot to my fellow graduate student Antoine Raux, who has been the perfect sounding board and research companion throughout my years at Carnegie Mellon University.

A number of other people have also helped significantly. I would like to thank Tim Paek and Greg Aist for many insightful discussions during my internships and later. Alan Black and Maxine Eskenazi have facilitated my work in the context of the Let's Go! Public system, and provided valuable feedback. Jack Mostow, Rich Stern, Mosur Ravishankar, Bob Frederking, Jahanzeb Sherwani, Satanjeev Banerjee, Stefanie Tomko, Ananlada Chotimongkol and the other faculty and students at the Sphinx lunches have made every Thursday at noon an opportunity to bounce ideas and a learning experience. Similarly, this work has benefited from my interactions with the students in the Dialogs on Dialogs student reading group. I would like to thank them all, and I would like to thank Alex Rudnicky for encouraging me to start this reading group and for his continued support of it and of all my other efforts. I would also like to thank Marian Boldea for his help and for giving me the rare chance to engage in dialog research as an undergraduate in Romania.

I would like to thank Julie Brick for her constant support, encouragements and help, and for putting up with me during my many deadlines. I would also like to thank Mihai and Diana Rotaru, Yan Karklin, Francisco Pereira, Maverick Woo, Chris Colohan, Marius Minea, Mihai and Raluca Budiu, for their friendship, advice and for keeping my spirits up. Last but definitely not least, I would like to thank my parents for everything they have done for me. I dedicate this effort to them.

Contents

List of Figures

List of Tables

PART I.

PRELIMINARIES

Chapter 1

Introduction

Despite recent advances in component technologies, conversational spoken language interfaces are still very brittle when confronted with understanding-errors. Most of these errors stem from limitations in current speech recognition technology, and as a result they appear across all domains and interaction types. Left unchecked, they exert a significant negative impact on the overall quality and success of the interactions. In this introductory chapter, we articulate a research program that aims to improve robustness in spoken language interfaces by creating the necessary mechanisms for seamlessly and efficiently recovering from errors. They key components are: (1) endowing spoken language interfaces with better error awareness, (2) developing and evaluating a rich repertoire of error recovery strategies, and (3) developing scalable, data-driven approaches for making error recovery decisions. This chapter outlines the contributions this dissertation brings to the proposed research program and provides a roadmap for the rest of this document.

1.1 Introduction

Over the recent years, advances in automatic speech recognition, as well as language understanding, generation, and speech synthesis have paved the way for the emergence of complex, task-oriented conversational spoken language interfaces. Here are a few examples: Jupiter (MIT) [142] provides up-to-date weather information over the telephone; CMU Communicator (CMU) [101] acts as a travel planning agent and can arrange multi-leg itineraries and make hotel and car reservations; TOOT (AT&T) [121] gives spoken access to train schedules; Presenter (Microsoft) [83] provides a continuous listening command and control interface to PowerPoint presentations; WITAS (Stanford) [66] provides a spoken language interface to an autonomous robotic helicopter; AdApt (KTH) [45] provides real-estate information in the Stockholm area; TRIPS (Rochester) [34] is a spoken-language enabled planning assistant. These are only a few; for a longer list, see [10].

Traditionally, the research community has focused on building spoken dialog systems[1] for

[1] In this document, we will use the terms **spoken dialog system** and **(conversational) spoken language interface** interchangeably. The first term has been traditionally used by a large number of researchers, and highlights the "system" nature of these artifacts and the fact that they rely on a large number of communicating subcomponents that perform different functions. The second term highlights the "interface" or "interactive media" aspects, and the fact that building good speech interfaces requires significant expertise in the field of human-computer interaction (see discussion in [46])

information access and command-and-control tasks. Current initiatives aim at the development of more sophisticated language-enabled interactive agents, such as personal assistants (e.g. CALO [21], LARRI [13]), taskable agents (e.g. Orion [59, 110]), interactive tutors (e.g. Paco [96]), call-routing systems (e.g. How May I Help You? [41]), embodied conversational agents (e.g. Rea [23], Olga [120], etc.) At the other end of the complexity spectrum lie simpler systems that have already transitioned into day-to-day use. United Airlines handles claims for missing and delayed luggage via their spoken dialog system called Simon. Newsweek magazine uses one for subscription renewal and cancellations. Various companies (e.g. Nuance, TellMe, BeVocal, HeyAnita, etc.) are successfully commercializing platforms and applications to deliver voice dialing, messaging and a large variety of telephone-based automatic customer care services.

The development of these systems has benefited from, but at the same time prompted, numerous advances in the component technologies. Speech recognition accuracy has improved over time, allowing spoken dialog systems to handle larger vocabularies and user populations. Dialog management paradigms, such as finite-state, form-filling, task or plan-based dialog have been proposed and empirically validated in the field. Novel text-to-speech synthesis techniques, such as unit-selection, have allowed the development of natural voices in limited domains. A certain consensus has been reached with respect to how to architecturally structure a spoken dialog system. Nevertheless, a lot of problems remain in need of better solutions. Automatic recognition and understanding of spontaneous speech are still far from perfect and as a result spoken language interfaces remain very brittle. Current solutions for system development do not transfer well across tasks and domains, and require considerable amounts of human effort and expertise. More complex aspects of human-computer interaction, such as multi-modality, multi-party conversations, embodied interfaces, fine-granularity timing and turn-taking are just beginning to be addressed.

1.2 The problem

Perhaps one of the most important and persistent problems in the development of spoken language interfaces is their lack of robustness when confronted with understanding-errors. Most of these errors stem from limitations in current speech recognition technology, and, as a result, they appear across all domains and interaction types. Automatic speech recognition is a difficult problem in general. In the context of spoken language interfaces, the difficulties are further exacerbated by the conditions under which these systems are meant to operate. In general, these systems are targeted to large user populations. As a consequence they have to accommodate large variations in speaking styles (i.e. accents, native, non-native speakers), as well as variations in the quality of the input lines (e.g. regular phone lines, cell-phones, close talking microphones) and in the amount and type of environmental noise. Additionally, they have to deal with various spontaneous speech phenomena like disfluencies, stutters, false starts, repairs, hesitations, filled pauses, and various other non-lexical acoustic events.

Under these circumstances, it is not surprising that typical word-error-rates in all but the most simple systems range from 15% to as high as 40% or even higher for non-native populations or under adverse noise conditions [15, 89]. Unless mediated by a better awareness and robust recovery mechanisms, speech recognition errors propagate to the higher levels of these systems and exert a considerable impact on the quality and success of the interactions.

1.2.1 An example

To make these problems more concrete, consider the sample conversation from Figure 1. This example was extracted from a corpus of dialogs collected with the CMU Communicator [101], a spoken dialog system that operates in the air travel planning domain.

In the first turn, the system asks for the departure city. Unfortunately, the user's response – 'Urbana Champaign' – is misrecognized by the system as *'OKAY IN THAT same PAY'*. The language understanding component fails to generate a semantic representation for this recognition result. The

1 S: What city are you leaving from?
2 U: *Urbana Champaign*
R: ***[OKAY IN THAT same PAY]***
3 S: Sorry I'm not sure I understood what you said. What city are you leaving from?
4 U: *Urbana Champaign*
R: ***[for minute since HEY]***
5 S: I'm still having trouble understanding you. To go on, I need you to answer the following question. What city are you leaving from?
6 U: *Chicago*
R: ***[CHICAGO]***
7 S: traveling from Chicago. Where would you like to go?
8 U: *Huntsville*
R: ***[SEOUL]***
9 S: traveling to Seoul. What day did you need to travel?
10 U: *no no I'm traveling to ... Birmingham*
R: ***[THE TRAVELING to berlin P_M]***
11 S: traveling in the afternoon. Okay what day would you be departing Chicago?
12 U: */uh/ the tenth of august*
R: ***[AT THE TENTH OF AUGUST]***
13 S: a flight tomorrow ... I have a Korean Airlines flight departing Chicago at 1:40pm, arrives Seoul at 5pm the next day. Do you want to take that?
14 U: *my destination is Birmingham*
R: ***[flight destination mr WEEKEND]***
15 S: traveling on Saturday, August 12th ... I have a flight departing Chicago at 1:40pm arrives Seoul ...

Figure 1. A sample conversation with the CMU Communicator [101] (S: marks system utterances; U: marks user utterances; R: marks recognition results; the words in lowercase in the recognition results are marked as unconfident by the speech recognizer and are ignored for language understanding purposes

system knows that the user said something, i.e. an utterance was detected, but has no useful information about the contents of this utterance. In this case, we say that a **non-understanding** has occurred. In turn 3, the system notifies the user that it did not understand the previous utterance, and repeats the question. The user responds again 'Urbana Champaign', but unfortunately his response is again misrecognized. This second recognition error led to another non-understanding. In the absence of a mechanism for diagnosing the source of the non-understandings, or of a richer set of error recovery strategies, the system persists in asking the same question again in turn 5. At this point, the user decides that the system is unlikely to correctly understand Urbana Champaign, and changes the departure to another nearby city – Chicago (turn 6).

Next, in turn 7, the system asks about the arrival city. The user responds 'Huntsville'. This city name was not in the system's vocabulary, and the recognition engine produces another erroneous result: *'Seoul'*. In this case, we say that a **misunderstanding** has occurred: the system has obtained a semantic interpretation of the user's utterance, but this interpretation is unfortunately incorrect. In the next turn (9), the system echoes back the value it heard (Seoul) to the user, and continues with the next question (what day did you need to travel?); the assumption is that, if the arrival city value was incorrect, the user would be able to detect this error and correct the system. This is indeed what happens in the dialog, but unfortunately the user's correction is also misrecognized by the system (turn 10). Without the ability to assess the reliability of the hypotheses generated by the recognizer and to update its beliefs based on subsequent user responses, the system takes this misunderstanding as fact, and continues the dialog using invalid information. Next, the system asks for the departure

date again (turn 11) and this time it correctly recognizes the user response (turn 12). However, the results presented to the user in turn 13 are affected by the previous misunderstandings. In turn 14, the user attempts again to correct the system, but to no avail.

1.2.2 Some statistics

Unfortunately, the sample conversation from Figure 1 is not just a contrived corner-case interaction carefully chosen to make a point. The example was indeed selected to illustrate a spectrum of different problems over a short number of turns. However, these types of breakdowns are not uncommon at all in any state-of-the-art spoken dialog system.

Here are some statistics. An analysis of the CMU Communicator corpus indicated that 32% of the utterances contain understanding-errors [22]. Similar results are reported in the literature for other spoken dialog systems [62, 68, 89, 121]. Research on semantic confidence annotation shows that typically 20 - 40% of the utterances addressed to a system are not understood correctly (see Table 1). Other work shows that approximately a quarter to one third of the utterances addressed to

System	Semantic error rate
CMU Communicator [101]	32%
CU Communicator [103]	27%
How May I Help You [130]	36%
Jupiter [49]	28%
SpeechActs [138]	25%

Table 1. Semantic error rates in current spoken dialog systems

a system are dedicated to correcting system mistakes: 32% in a Dutch spoken dialog system that provides train time-table information [62]; 29% in TOOT [121]. Due to changes in speaking style (e.g. frustration, hyper-articulation), these corrections are themselves two to three times more likely to be misrecognized than other utterances [68, 121].

The relatively large numbers of understanding-errors and corresponding user corrections have a significant impact on the overall quality and success of the interactions. In the CMU Communicator, the 32% understanding-error rate results in 66% of the sessions having at least one understanding-error. In 40% of these cases (26% of the total number of sessions), the end result was complete breakdown in interaction and failure to achieve the task. The users managed to get the interaction back on track in only 60% of the sessions containing misunderstandings. In a similar error analysis study, Shin and Narayanan [113] found 235 error segments in 141 dialogs in the air-travel planning domain; this equates to an average of 1.66 error segments per session. 22% of the error segments (37% of the sessions) led to a complete breakdown of the interaction.

The effect of speech recognition errors on overall performance is illustrated in Figure 2. This

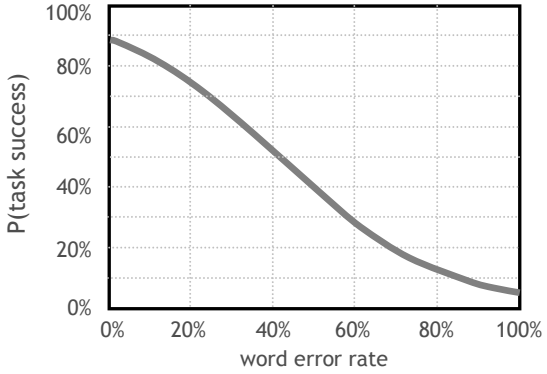


Figure 2. Probability of task success as a function of average word-error-rate in a deployed spoken dialog system

plot shows the probability of task success as a function of average word-error-rate in a deployed spoken dialog system that can assist users in making conference room reservations. As this figure shows, there is a sharp drop in the probability of task success in the 15-40% word-error-rate range. In addition, repeated studies [128, 129] have found a strong inverse correlation between word-error-rate and user satisfaction.

The statistics discussed above, as well as large amounts of anecdotal evidence show that these problems occur in significant numbers across most domains and interaction types.

1.2.3 Factors and limitations underlying interaction breakdowns

Today, the speech recognition process is the primary source of errors in spoken language interfaces. Would speech recognizers produce perfect results, most problems would disappear[2]. We would mostly be left with the task of managing the dialog under assumed perfect inputs, a problem for which various approaches have been proposed and are evolving in the community. In reality, automatic recognition of spontaneous speech is still imperfect at best. In the example from Figure 2, the recognition errors were triggered by the use of out-of-vocabulary words; this is however just one of a number of factors that can trigger such errors. Small changes in the environment, microphone or telephone line quality can seriously impair recognition performance. Since spoken dialog systems are generally targeted to large user populations, speaker variability represents another major concern. If we add to this mix the disfluencies which characterize spoken language, the recognition word-error-rates can jump from a typical 8-10% for read speech to as high as 30%, or even higher, for spontaneous speech.

The impact of recognition errors on overall dialog performance is very pronounced because systems are not well prepared to handle the resulting uncertainties. Two important deficiencies are: (1) shortcomings in the ability to accurately detect and diagnose problems, and (2) a lack of sufficient and effective error recovery strategies. Moreover, a principled yet practical computational framework for making error handling decisions (3) is still missing. In the absence of these three core competencies, speech recognition errors will continue to exert a strong negative impact on overall performance, and to severely limit the naturalness of the interaction and the complexity of tasks that can be addressed. In the reminder of this section, we discuss each of the three limiting factors outlined above in more detail.

§ Error detection and diagnosis

A first important deficiency that can be observed in many current spoken dialog systems is their inability to accurately identify and diagnose errors. Left unchecked, speech recognition errors can lead to two types of understanding-errors: **non-understandings** and **misunderstandings**. A **non-understanding** occurs when the system fails to acquire any useful information from the user's turn. For instance, if the parsing component lacks robustness, then even the smallest recognition error can ruin the whole language understanding process. In such cases no meaningful semantic representation of the input can be obtained, and a non-understanding occurs. In contrast, a **misunderstanding** occurs when the system extracts some information from the user's turn, but that information is incorrect. This generally happens when the language understanding layer generates a plausible semantic representation, which happens to match the current discourse context, but which does not correspond to the user's expressed intent. In Chapter 2, we will discuss in more detail these two types of understanding-errors and investigate their sources and impact on overall performance.

In the absence of a mechanism for assessing the reliability of their inputs or for updating their beliefs throughout the conversation, systems will take misunderstandings as fact and will act

[2] Other sources of understanding errors exist besides the recognition process. Parsing (i.e. grammar coverage) errors can lead to misunderstandings or non-understandings as well. High-level interpretation, reference resolution, and intention recognition can also create similar problems. Nevertheless, in most spoken dialog systems the largest proportion of errors stem from the speech recognition process – see [20, 22], as well as our empirical analysis from Chapter 6.

based on invalid information. We have seen this happening in turns 8, 10 and 14 of the example from Figure 1. For instance in turn 8 the system "heard" that the user had said the arrival city was Seoul. Ideally, we would like the system to be able to detect immediately that *'Seoul'* is an uncertain recognition result. In the next turn (10), the user tried to correct the value. While this response was also affected by recognition errors, the garbled recognition result should have further increased the system's skepticism about the correctness of Seoul. Lacking the ability to accurately update its beliefs based on evidence from subsequent user responses, the system persisted in the same error.

In contrast, when a non-understanding occurs no false information is incorporated into the system. However, the situation is not much better. Without a diagnosis process that can indicate the likely sources of the non-understanding, the system's follow-up options are limited and uninformed. In the given example they amounted to asking the question again and "hoping for better luck next time" (turns 3 and 5). A system that can detect the possible source of the error (was it an out-of-vocabulary word? a transient noise? does the user speak with a pronounced accent? does the user not know what to say? etc.) could make more informed decisions, and therefore be more likely to set the dialog back on track. The ability to assess how well the interaction is proceeding within a given turn, a certain discourse segment, or over the whole conversation, as well as to diagnose potential problems would provide a better basis for making error recovery decisions.

§ Error recovery strategies

A second important limitation that contributes to interaction breakdowns is the lack of effective conversational strategies for preventing errors and recovering from them. When faced with understanding-errors, humans use a large variety of strategies to set the conversation back on track [115, 141]: confirmations, disambiguations, repeating information, checking context, trying alternative plans to achieve the same dialog goal, even guessing or ignoring unreliable information. More strategies can be envisioned in the context of a task-oriented human-machine conversation: lexically entraining the users, providing help, falling back to a stricter initiative and constraining the input language, relying on alternate input modalities such as DTMF (touch-tone) or pen-based input, etc. Only a few of these strategies have been thoroughly investigated and are consistently used in today's spoken dialog systems. Our understanding of the advantages, disadvantages and relative trade-offs between these strategies is fragmentary at best.

§ Error recovery policies

The third missing component is a practical computational framework for making the error handling decisions. The problem of error handling is often regarded as an add-on, and the large majority of systems use handcrafted heuristic decision rules to engage in a small number of error recovery strategies. This approach lacks a solid basis. The technique cannot be extended to handle a larger number of strategies, since the trade-offs that need to be solved become ever more complex. More systematic approaches, based on Bayesian decision theory and reinforcement learning, have been proposed but so far they lack scalability are not applicable in large, practical spoken dialog systems.

§ Other factors

The factors discussed above are not the only contributors to the brittleness observed in today's spoken dialog systems. Other factors, such as inconsistency and poor design of the system's actions at the task level, deficiencies in the user's knowledge about the system's functionality, lack of basic conversational skills on the system side (i.e. timing, turn-taking, handling, repeat, etc.), and lack of support for user-initiated repairs can also lead to interaction breakdowns.

1.3 Towards increased robustness in spoken dialog systems

The discussion from the previous section highlights two different pathways for increasing the robustness of spoken language interfaces. One approach is to increase the accuracy of the speech recognition and language understanding process, in an effort to eliminate the errors altogether. A differ-

ent approach is to assume some errors will always be present and create the mechanisms for recovering from them through conversation (i.e. by acting robustly at the dialog management level.)

Reducing the speech recognition error rates would certainly lead to corresponding improvements in the performance. Not surprisingly, the problem has received a great deal of attention. Numerous efforts are directed at improving the baseline speech recognition performance: more robust representations, more sophisticated statistical models, better search, acoustic- and language-modeling techniques. The typical results so far have been consistent but small incremental improvements. In contrast, the most effective and widely applied technique for improving recognition accuracy in spoken dialog systems is remarkably unsophisticated: constrain the input language. By imposing limits on what users can say at any given point in the conversation, the recognition problem is simplified and the decoding process becomes less prone to errors. A widely used technique is to build systems that always keep the initiative, and constrain user responses based on the current focus of the conversation. Other techniques, such as interpolating state-specific and generic language models [43, 137], or backing off from grammar-based to SLM-based recognition, can regain some of the lost flexibility while still improving recognition performance. The Universal Speech Interface [99, 122, 123] is another approach, still aimed at alleviating the recognition problems. In this case the constraints are imposed on the form rather than the contents of the user's language. Users are trained (or entrained) to speak in a stylized fashion, within the bounds of a well-defined universal grammar. In the end, all these techniques trade off naturalness and flexibility for accuracy. They are applicable in a number of relatively simple domains and interaction types, and can indeed lead to increased robustness. However, these techniques do not provide a suitable solution if the goal is to build natural spoken language interfaces that operate in complex domains.

The alternative approach is to assume that the recognition process will always be unreliable, and compensate by creating the mechanisms for acting robustly at the conversation level. In this case, we need to address the limitations described in the previous section. Three ingredients are required. First, we need to endow spoken language interfaces with the ability to **detect and diagnose errors (1)**, preferably as soon as they happen. Second, we need to endow these systems with a rich repertoire of **error recovery strategies[3] (2)**. Last, we need to develop techniques that allow these systems to optimally choose between these strategies at runtime (e.g. when should the system ask the user to repeat? When should it ask the user to rephrase? etc.). In other words we need to endow systems will good **error recovery policies (3)**.

This second approach has a number of advantages. While incremental improvements in speech recognition performance continue to be made, the demands also increase, as the interests shift towards ever more complex domains and interaction types. This visible trend predicts that recognition performance will not satisfactorily match the tasks at hand in the near future. By default, spoken language interfaces will remain either brittle or significantly constrained. In contrast, systems that can handle the uncertainties at a high level can more safely relax the language constraints, and allow increasingly natural interactions over complex domains, even in the presence of imperfect recognition. The approach bears similarities to the ways humans handle uncertainties in communication: after all, human speech recognition capabilities are not perfect either, but we are able to repair conversations and reestablish mutual ground when misunderstandings occur. Although the recognition process is a major source of problems in spoken dialog systems, it is not the only such source. Errors can be introduced by other components such as language understanding, reference resolution and intention recognition. Increasing the robustness at the dialog management level will also allow spoken dialog systems to accommodate other sources of uncertainty.

The two approaches briefly outlined above – improving recognition accuracy and recovering from errors through conversation – do not stand in opposition. Clearly, a combined effort would

[3] by **error recovery strategy** we will denote a one-step conversational action that the system might take to set the conversation back on track, such as asking the user to rephrase, asking the user to speak less loud, confirming a certain concept value, etc.

lead to the best results. The research program described in this document focuses on the second approach: it aims to construct the mechanisms for seamlessly and effectively recovering from errors through conversation.

1.4 Thesis goals

The high-level goal of this dissertation is therefore to **increase the robustness of task-oriented spoken language interfaces by creating the necessary mechanisms for recovering from errors through conversation**.

We propose to accomplish this high-level goal by:

(1) endowing spoken language interfaces with better **error awareness**;

(2) constructing and evaluating a rich repertoire of **error recovery strategies**;

(3) developing an adaptive, scalable, data-driven approach for making error recovery decisions, i.e. an **error recovery policy.**

Together with the two types of understanding-errors we have previously identified (misunderstandings and non-understandings) these three components define the underlying coordinates for the research program described in this document, as illustrated in Figure 3.

We seek task-independent, adaptive, and scalable solutions for these problems. First, we seek **task-independent** solutions: the error handling mechanisms should ideally be decoupled from the particularities of the actual dialog task performed by the system. This separation can favor the reuse of the proposed solutions across different systems operating in different domains, and in the process lessen the system authoring effort. Furthermore, the decoupling would ensure uniformity and consistency in error handling behaviors both within and across tasks.

Secondly, we seek **adaptive** solutions. Spoken language interfaces operate under a large variety of conditions: different performance in the underlying speech recognition and language understanding components, different and changing user populations, different qualities of the input lines, different costs for various types of errors, etc. We seek data-driven, learning-based solutions that compensate for these differences by adapting to the characteristics of the domain in which the system operates. Moreover, in the ideal case systems should learn online, from their own experiences, without requiring extensive supervision from their developers.

Last but not least, we seek **scalable** solutions. Previous learning-based approaches in dialog

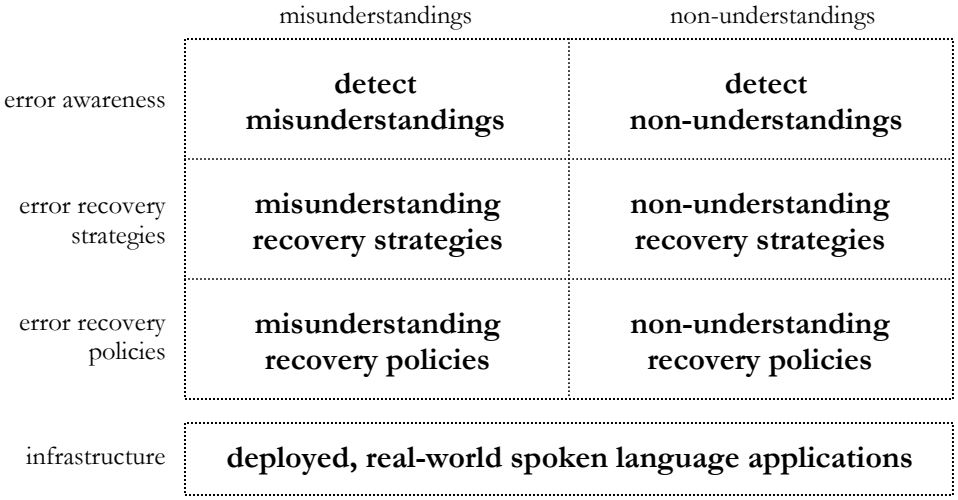


Figure 3. A six-component research program for increased robustness in spoken language interfaces

control have shown some success in limited domains, but do not scale to large, real-world spoken language interfaces. The computational costs for learning and acting should grow at most linearly with the size of the dialog task, making the solution applicable in real-world systems operating with larger, more complex tasks.

To validate these properties, we will empirically evaluate the proposed solutions in the context of a number of real-world, deployed spoken language applications, built upon of the same dialog management and error handling infrastructure.

The dissertation work described in this document brings a number of important contributions within the problem space described above. At the same time, it does not complete the entire research program we have outlined. Rather, it is best viewed as a concerted effort at advancing the state-of-the-art in a number of these areas, while also raising a number of additional scientific and technical questions.

1.5 Brief overview of current solutions

The six sub-problems we have outlined in Figure 3 have already received a fair amount of attention. In this section, we give a brief overview of typical current solutions – Figure 4 provides a graphical summary. Our goal here is not to provide an exhaustive review of error handling research, but rather to outline some of the common solutions and describe the larger context for the work undertaken in this dissertation. We will review the relevant literature and related research for each of the issues we address later on, in the corresponding chapters.

Current spoken language interfaces generally rely on confidence scores to **detect misunderstandings**. These scores are computed automatically by a confidence annotation model and are meant to reflect the degree of accuracy of the decoding or language understanding process. Typically, supervised learning techniques are used to build confidence annotation models: a corpus of utterances is manually labeled, a set of potentially relevant features is identified, and a classifier is trained to predict whether or not a given utterance was correctly understood by the system. Although confidence scores are not perfectly accurate, they can provide an initial assessment of the reliability of the information received from the recognizer. Hence, they can be used to detect potential misunderstandings: a low confidence score means an increased likelihood of misunderstanding and a high confidence score means a decreased likelihood of misunderstanding.

The most common[4] **strategies for recovering from misunderstandings** are explicit and

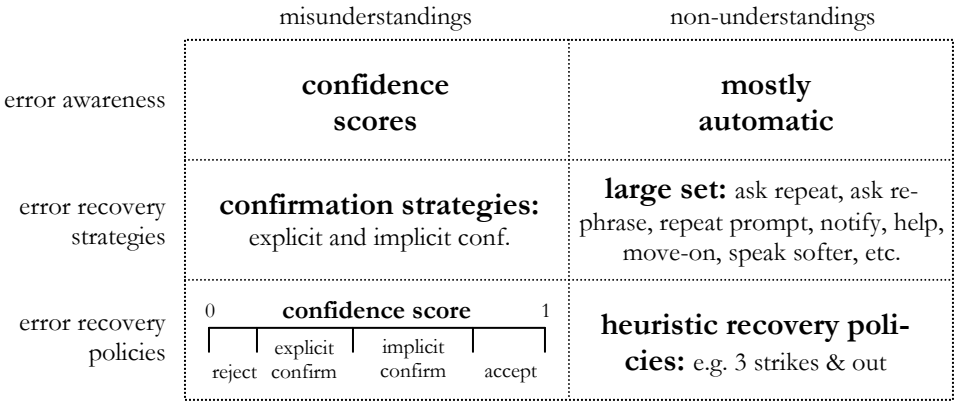


Figure 4. Brief overview of typical current solutions for error handling

[4] Although other strategies such as disambiguation might be used (e.g. "Did you say Boston or Austin?"), they are in general rare because the increased complexity of subsequent user responses can lead to significant recognition and understanding problems.

implicit confirmation. In an explicit confirmation the system asks a yes/no question to validate a piece of information (e.g. "Did you say you were flying from Boston?"). In an implicit confirmation, the system echoes back the value it wants to confirm to the user, and then continues with the next dialog contribution (e.g. "flying from Boston … when would you like to leave?"). The assumption is that, if there was a misunderstanding, the user will detect the error and interject a correction. Alternatively, if the value is correct, the user will implicitly signal that to the system by simply responding to the next question; the dialog will advance normally towards its goals.

The **policies for engaging the misunderstanding recovery strategies** are typically based on a predetermined set of confidence thresholds, also known as confirmation thresholds. If the confidence score for an utterance[5] is very high, the system accepts the utterance and considers it grounded. If the confidence score is medium-high, the system might engage in an implicit confirmation; chances are the value is correct and the dialog will advance normally. Alternatively, if the confidence score is medium-low, the system will engage in an explicit confirmation: in this case, the increased likelihood of misunderstanding justifies the cost of taking an extra dialog turn to validate that information. Finally, if the confidence score is very low, given the very high likelihood of misunderstanding, the system might decide to reject the utterance altogether.

For **non-understandings, detection** is in general a trivial task. By definition, the system automatically knows when non-understandings occur: the user takes a turn (a speech signal is detected), but no meaningful information can be extracted from the recognized hypothesis. There is one special case – rejection non-understandings – in which detection does pose questions. A rejection non-understanding occurs when the system decides to reject the utterance because of a low confidence score, even though it could extract information from the user's turn. In effect, the system creates a (rejection) non-understanding to avoid a potential misunderstanding. An interesting question in this case (discussed later in this work) is how can we set the rejection threshold in a principled manner? Commonly encountered solutions to this problem rely on simple rules of thumb regarding the relative cost of false-rejections.

A relatively large set of **non-understanding recovery strategies** have been proposed and used in various spoken language interfaces. When a non-understanding occurs, a number of strategies can be used to attempt a repair. For instance, the system may ask the user to repeat; it may ask the user to rephrase; it may repeat its previous question; it may notify the user that a non-understanding has occurred; it may give various levels of help; it may ask users to speak softer, louder, with fewer or longer utterances, etc.; it may even ignore the current non-understanding and try to advance the dialog in a different manner, by moving on to a different question or using an alternative dialog plan. Although a large number of strategies have been proposed in different systems, in most cases only a limited subset of such strategies is used.

In contrast to misunderstandings, the relative trade-offs between non-understanding recovery strategies are not very well understood. As a consequence, most systems use very simple, heuristic **policies for engaging the non-understanding recovery strategies** in conjunction with a small set of recovery strategies. A commonly encountered policy is to cycle through three different recovery strategies on consecutive non-understandings and to transfer the user to an operator after three consecutive failures. This is also known as the "three strikes and you're out" approach [5].

To summarize, for misunderstandings, detection is the key issue. If a system is able to accurately assess the reliability of the information it operates with, it can then use various confirmation strategies (e.g. explicit or implicit confirmation) to verify information when needed. These strategies have been previously investigated [62, 132] and they are fairly well understood. Furthermore, while some tuning is involved, a simple threshold-based mechanism is generally sufficient for engaging these strategies. In contrast, for non-understandings, detection is in general a simple task: the system automatically knows when non-understandings occur because a user input is detected but no meaningful semantic interpretation can be constructed for the corresponding recognition result. At the

[5] policies for recovering from misunderstandings can operate either at the utterance or at the concept level.

same time, the set of strategies that can be used to recover is much larger, and the relative trade-offs between these strategies are less well understood. As a result, the construction of policies for recovering from non-understanding is perhaps the most challenging task for this type of errors.

1.6 Thesis contributions

This dissertation brings a number of contributions in the problem space described above. For a quick overview, the complete list of contributions (numbered C1 through C10) is outlined in Figure 5. For indexing purposes, the number in parentheses in Figure 5 shows the chapter in which that contribution is described in detail. The primary contributions and main thrust of this work centers on the two more challenging problems we have identified in the previous section: detection of misunderstandings and recovery policies for non-understandings. At the same time, we make several other contributions with respect to problems such as detection of non-understandings, error recovery strategies (both for misunderstandings and non-understandings), as well as error handling infrastructure. Below, we provide a synopsis of these contributions.

§ Detection of misunderstandings

This dissertation brings three important contributions with respect to the issue of detecting misun-

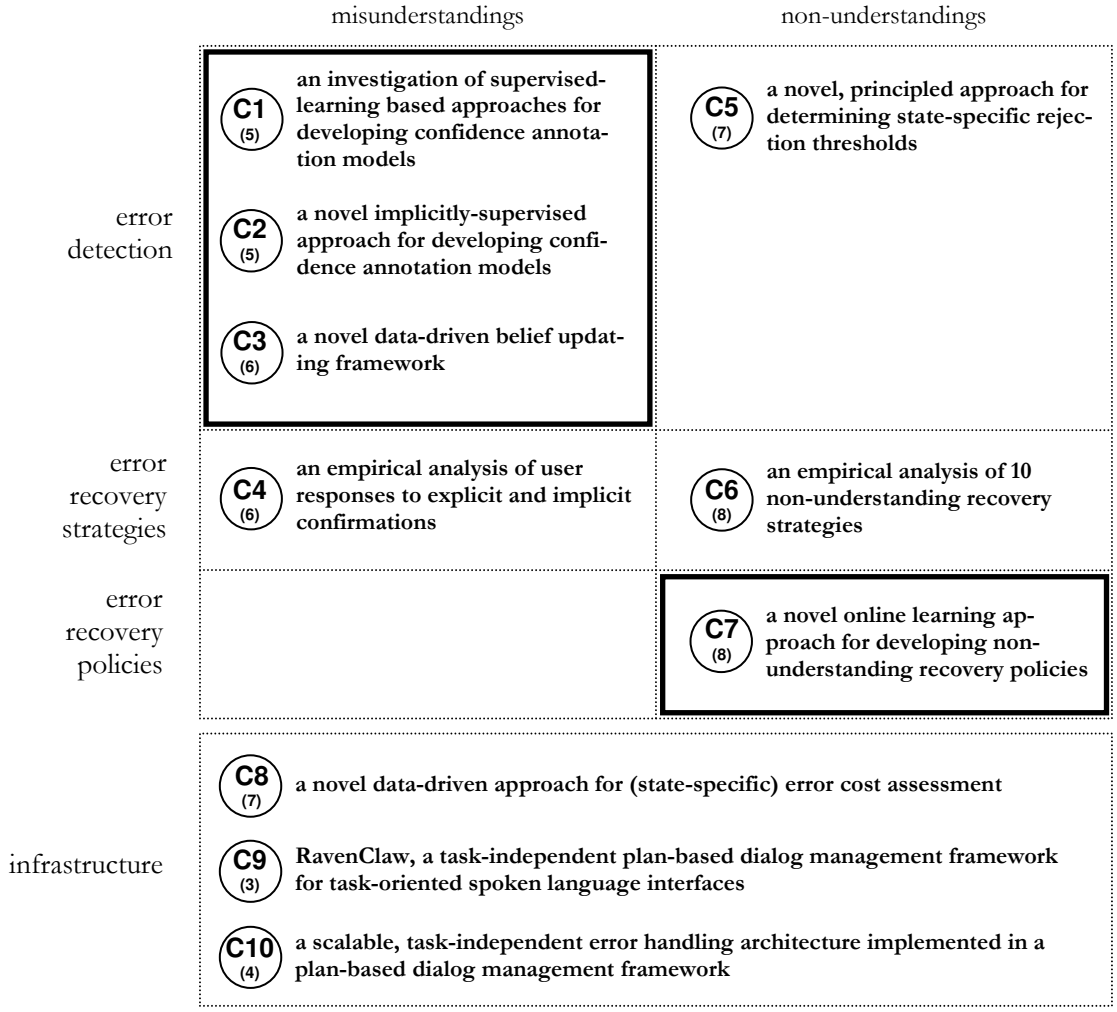


Figure 5. A summary of contributions; for indexing purposes, the number in parentheses indicates the chapter in which each contribution is described

derstandings.

First, we have performed a thorough **investigation of four supervised-learning techniques for developing confidence annotation models (C1)**. While supervised-learning is commonly used for building confidence annotation models, we have focused our attention on a number of issues less thoroughly investigated in the literature. These include the appropriateness of various evaluation metrics for confidence annotation, the sample efficiency of various supervised learning techniques and the transferability of confidence annotation models across domains. We advocate the use of proper probabilistic scoring rules for evaluating confidence annotation performance and show empirically that relying solely on classification-error (a commonly used approach) is insufficient. Empirical results show that, when enough data is available for training, models constructed using different supervised learning techniques perform similarly. No single technique is most appropriate for this task; rather, the power is in the feature set. However, when only little training data is available, logistic regression models significantly outperform other supervised learning techniques. Finally, we also investigate how well confidence annotation models transfer across domains. Results indicate that, while some models transfer well, this is not always the case. We propose a simple post-transfer calibration procedure that uses a small amount of labeled data in the target domain and generally improves the performance of the transferred model.

Second, we have proposed a **novel, implicitly-supervised approach for learning confidence annotation models (C2)**. Traditional supervised-learning techniques require a pre-existing corpus of labeled instances. This is often costly and labor intensive to acquire; furthermore supervised learning favors a batch approach that leads to fixed, un-adaptive solutions. We have proposed a novel approach (dubbed implicitly-supervised learning) in which no developer supervision is required. Instead, the system automatically extracts the required supervision signal online, from a correction pattern that naturally occurs in the conversation. In effect, the system can learn throughout its lifetime, from its own experience with the users. Experimental results in two different domains indicate that the proposed approach can attain 80% of the performance of a traditional, fully-supervised model. We believe this novel learning paradigm can be applied in a number of other problems in spoken dialog (as well as other interactive systems) and constitutes an important step towards building autonomously self-improving systems.

The third and perhaps central contribution this dissertation makes with respect to the problem of detecting misunderstandings is the development of **a scalable data-driven belief updating framework (C3)**. We have seen that systems typically rely on confidence scores to form an initial assessment of the reliability of the information obtained from the speech recognizer. Ideally, spoken language interfaces should continuously monitor and improve the accuracy of their beliefs by integrating evidence across multiple turns in a conversation. In this work, we formalize this belief updating problem, and propose a scalable, supervised-learning based solution. An empirical evaluation with a deployed spoken dialog system shows that the proposed approach constructs significantly more accurate beliefs than previous heuristic solutions and leads to large gains in both the effectiveness and the efficiency of the interaction.

§ Misunderstandings: recovery strategies

Dialog systems typically rely on confirmation strategies to recover from potential misunderstandings. To gain a better understanding of these strategies, we have performed an **empirical analysis of user responses to explicit and implicit confirmations (C4)**. We followed a methodology previously used in a similar study by Krahmer et al. [63]. Our results corroborate their previous observations in a different domain. They indicate that user responses to confirmation strategies cover a wide language-spectrum, especially after implicit confirmations, and especially when the information to be confirmed is incorrect. Furthermore, we have found that users interact strategically with the system: often they will not correct the system following a confirmation strategy, unless the correction is essential for the task at hand. These observations add to an existing body of knowledge regarding the functioning of these strategies, and shed more light on the challenges we face in developing accurate misunderstanding detection and belief updating mechanisms.

§ **Misunderstandings: recovery policies**

The work described in this dissertation does not offer a direct contribution to the area of developing misunderstanding recovery policies. We shall note however that, the data-driven error-cost assessment methodology developed in contribution C8 (see below) could be extended to infer costs for various confirmation actions. This methodology can therefore provide a more principled basis for tuning the confirmation thresholds that typically define misunderstanding recovery policies.

§ **Non-understandings: detection**

As we have previously noted, detection of non-understandings is in most cases trivial. There is one interesting exception – that of rejection non-understandings, i.e. situations in which the system purposefully rejects an utterance because of a low confidence score. An interesting question in this case is what should the rejection threshold be? In this work, we propose **a principled approach for determining state-specific rejection thresholds (C5)**. The approach relies on a data-driven methodology for assessing the costs of errors, described in contribution C8. Experimental results confirm the intuition that different rejection thresholds should be used at different points in the dialog, corroborating previous anecdotal evidence from observing the system.

§ **Non-understandings: recovery strategies**

The next contribution is an in-depth **empirical investigation of 10 non-understanding recovery strategies (C6)**. The analysis focused primarily on identifying the relationships between each strategy and subsequent user responses and determining which user behaviors are more likely to lead to successful recovery. In addition, we investigated the relative performance of various non-understanding recovery strategies, when engaged in an uninformed manner and when engaged using a "smarter" policy, implemented by a human operator in a wizard-of-oz setup. The results add to an existing body of knowledge about the relative advantages and disadvantages of these recovery strategies, and highlight the importance of good recovery policies.

§ **Non-understandings: recovery policies**

Developing good recovery policies is perhaps the most challenging task for non-understandings. This is especially difficult when the system is equipped with a large set of recovery strategies. In this dissertation, we have proposed and evaluated **a novel online-learning based approach for developing non-understanding recovery policies (C7)**. The proposed approach consists of two steps: first, we construct runtime estimates for the likelihood of success of each recovery strategy, together with confidence bounds for those estimates. Then, we use these estimates to construct a policy online, while balancing the system's exploration and exploitation goals. Initial experiments in a deployed spoken dialog system indicate that the proposed approach produces statistically significant improvements in the average non-understanding recovery rate.

§ **Infrastructure and other contributions**

In addition, this dissertation brings a number of other, cross-cutting contributions to the field of error handling in conversational spoken language interfaces.

One such contribution is **a novel data-driven approach for error-cost assessment (C8)**. The method relates the number of errors (of different types and at different points in the dialog) to a chosen global dialog performance metric. In the process, it allows us to infer the costs of these errors from data. Error-costs computed using this methodology in conjunction with data from a deployed dialog system corroborate our intuitions and our prior experience with this system. These costs are very useful for adjusting error handling behaviors. For instance, in contribution C5 we show how these costs can be used to determine state-specific rejection thresholds in a principled manner. More generally, we believe the proposed methodology can be applied to infer costs for various confirmation actions; in turn, these costs can be used to infer confirmation thresholds and construct better misunderstanding recovery policies.

The experimental platform for evaluating the solutions proposed in this dissertation consists of a number of real-world, deployed spoken language interfaces. These systems were constructed

using **RavenClaw, a plan-based, task-independent dialog management framework (C9)**. Apart from supporting the error handling research program that constitutes the main focus of this dissertation, RavenClaw represents an important contribution to the field of plan-based dialog management. The framework enforces a clean separation between the domain-specific and domain-dependent aspects of the dialog control logic, and in the process significantly lessens the system authoring effort. To date, RavenClaw has been used to build and successfully deploy a dozen spoken dialog systems spanning different domains and interaction-types [1, 13, 14, 47, 89, 90]. Together with these systems, RavenClaw provides a robust basis for other research projects addressing issues such as dialog management, timing and turn-taking [86] and multi-participant conversation [47].

To support the error handling research described in this dissertation, we have developed **a scalable, task-independent error handling architecture (C10)** in the context of the plan-based RavenClaw dialog management framework. The proposed error handling architecture decouples the set of error handling strategies, as well as the mechanisms used for engaging them, from the domain-specific aspects of the dialog control logic. This significantly lessens the development effort. System authors describe the domain-specific dialog control logic under the assumption that inputs to the system will always be perfect; the responsibility for ensuring that the system uses valid information and that the conversation advances normally towards its goals is delegated to the error handling mechanisms in the dialog engine. This decoupling facilitates the reuse of error recovery strategies and policies across domains and ensures uniformity and consistency in behavior both within and across domains. Although encapsulated approaches to error handling have been previously developed in different contexts, to our knowledge this is the first systematic, task-decoupled error handling architecture developed in the context of a complex, plan-based dialog management framework.

§ Future directions

The work described in this dissertation represents a concerted effort in the area of error recovery in conversational spoken language interfaces, within the confines of the research program we have previously outlined. At the same time, the proposed research program is by no means complete. Each of the contributions summarized above leaves a number of specific, follow-up questions. For instance: how can we generalize error detection models across different dialog domains? Can we further improve the performance of the proposed belief updating models (C3) by using information from an n-best list? Can we use the proposed data-driven error cost assessment models (C8) to develop misunderstanding recovery policies in a principled manner? Does the proposed online method for learning non-understanding recovery policy (C7) scale-up to even larger sets of recovery strategies?

In addition, this work also raises a number of more general research questions. For instance, we believe that the implicitly-supervised learning paradigm proposed in contribution C2 is applicable in a number of other learning problems. We conjecture that this paradigm can enable significant autonomous learning both in spoken dialog systems and in the larger class of interactive systems in general. The confidence annotation experiments we have conducted using this methodology (C2) represent only the first step towards understanding the properties, advantages and limitations of this approach. A number of important research questions remain to be answered: how can systems effectively exploit naturally-ocurring interaction patterns to acquire knowledge and improve their models online, through interaction? How can systems create such learning opportunities without having a negative impact on the interaction? How can systems automatically identify novel knowledge-producing patterns in the interaction and thereby increase the range of learning opportunities?

Another cross-cutting question regards the locality of the error handling process. In the work described in this dissertation, we have made the tacit assumption that error handling can be modeled as a (mostly) local process, and can be decoupled from the specific dialog task that the system implements. This assumption facilitates the development of scalable, portable and reusable solutions. At the same time, it introduces a number of limitations and drawbacks (we shall discuss these in more detail later.) In future work, it would be interesting to investigate more closely the impact of this assumption on the proposed solutions, both in terms of scalability and performance gains; an inherent trade-off between the two exists.

1.7 A reader's guide

This dissertation is organized into five parts.

Part I contains preliminaries. After this introductory chapter, in Chapter 2 we discuss in more detail the two types of understanding-errors that commonly affect spoken language interfaces: misunderstandings and non-understandings. We investigate the main sources of these errors. Then, we introduce a data-driven error-cost assessment method (8), and use it to assess the impact of these errors on global dialog performance.

Part II describes the infrastructure and experimental platforms used in the error handling research conducted in the rest of the dissertation. In Chapter 3 we outline the internals of the RavenClaw dialog management framework (9), and describe the various spoken language interfaces that have been developed using this framework. Next, in Chapter 4, we discuss the task-decoupled error handling architecture (10) in the RavenClaw dialog management framework, and explain how the rest of the work described in this dissertation fits into this architecture.

Part III deals with misunderstandings. It contains two chapters. In the first one, Chapter 5, we focus on the problem of turn level detection of misunderstandings (a.k.a. the problem of semantic confidence annotation.) We describe an in-depth investigation of four supervised-learning techniques for this task (1) in the first part of this chapter; then, in the second part, we discuss a novel, implicitly-supervised learning approach for the same problem (2). Next, in Chapter 6, we introduce and formalize the problem of updating beliefs across multiple turns in spoken language interfaces. We propose a scalable data-driven solution for this problem (3), and discuss empirical results using the proposed approach in a deployed spoken dialog system. In order to better portray the challenges we face in the belief updating task, we also report in this chapter results from an empirical analysis of user responses following two strategies used to recover from misunderstandings: explicit and implicit confirmation (4).

Part IV deals with non-understandings. The topic for Chapter 7 is detection of (rejection) non-understandings. More specifically, we propose a principled approach for determining state-specific rejection thresholds (5). We extend the data-driven error-cost assessment method introduced in Chapter 2 to include state distinctions (8). In Chapter 8 we focus our attention on non-understanding recovery strategies and policies. In the first part of this chapter we present an in-depth investigation of ten non-understanding recovery strategies (6), focused on understanding the relationship between each strategy and follow-up user responses. We then propose and evaluate a novel, online approach for learning non-understanding recovery policies over a large set of recovery strategies (7).

Part V summarizes the contributions and presents directions for future research. The dissertation document ends with a conclusion chapter, in which we briefly summarize the main contributions of this work, the lessons learned, and we outline a number of interesting directions for future research.

Chapter 2

Understanding-errors in spoken language interfaces

In the previous chapter we have introduced two types of understanding-errors that commonly affect spoken language interfaces: misunderstandings and non-understandings. In this chapter we analyze these errors in more detail. We begin by defining the terms misunderstanding and non-understanding more precisely in the first section. Then, in the second and third sections of this chapter we investigate the main sources of these errors and empirically assess their impact on global dialog performance.

2.1 Misunderstandings and non-understandings

Left unchecked, speech recognition errors can lead to two types of understanding-errors in spoken language interfaces: misunderstandings and non-understandings. In this section we will take a closer look at these two types of errors. We begin by describing a typical input processing architecture in a spoken language interface. Next, we provide precise definitions for misunderstanding and non-understanding errors and we introduce a number of related issues, such as confidence scores, rejections, and the misunderstanding / non-understanding trade-off. Then, in the next section, 2.2, we analyze the main sources of these understanding-errors. Finally, in section 2.3 we evaluate their impact on overall dialog performance and in section 2.4 we summarize the results we have found and present concluding remarks.

2.1.1 Input processing in spoken language interfaces

Spoken language interfaces typically process the user input in several consecutive stages. A classical pipelined input processing architecture is depicted in Figure 6. The main components are a speech recognition component, a language understanding component and the dialog manager.

To illustrate these processing stages, imagine the following exchange with a spoken dialog system that performs conference room reservations: the system asks "At what time do you need the room?", and the user responds *"until two p.m."*. The audio signal captured through a microphone (or telephone, etc.) serves as input to the speech recognition component. The recognizer decodes this audio signal and generates a recognition hypothesis for the user's response (`"two p.m."` in this case). Typically, speech recognition engines use end-pointers to perform speech and silence detection and

```
1  S: Until what time do you need the room?
2  U: until two p.m.
```

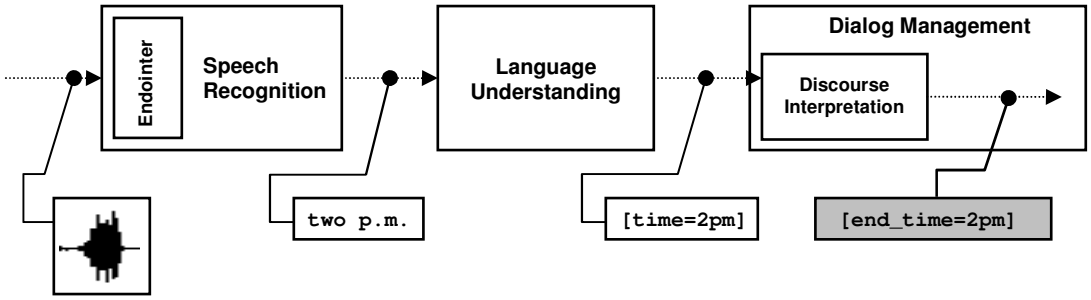


Figure 6. Typical input processing line in a spoken language interface

to segment the incoming audio signal into individual user utterances.

The recognition hypothesis is then forwarded to a language understanding component that constructs a semantic representation from the recognition result. Typically, either a frame (consisting of an attribute-value list and potentially an identifier) [84] or a logical form [33] is used to capture the semantics of the user's response. In the example from Figure 6, the semantic representation of the user input is an attribute-value combination: `[time=2pm]`.

The semantic representation is then forwarded to the dialog manager, where it will be interpreted in the larger discourse context. For instance, in our example, the dialog manager knows that it had previously asked for the end time for the reservation, and consequently the `[time=2pm]` semantic input corresponds to `[end_time=2pm]` at the discourse-level. If the same answer had been received when the system asked for the start time, the discourse-level interpretation would have been different, i.e. `[start_time=2pm]`.

The input processing line described above is typical for most spoken language interfaces. Nevertheless, differences might exist across various systems. Some systems use push-to-talk solutions to avoid end-pointing problems (e.g. systems that operate in open, noisy environments, or where no close-talking microphones are available). In these cases an end-pointer is not required. Some systems might use context free grammars instead of statistical n-gram language models. The same grammar that is used for recognition can in this case also be used for language understanding. The discussion that follows in the rest of this chapter as well as the rest of the work in this dissertation assumes the input processing architecture illustrated in Figure 6. However, most conclusions remain valid and most results remain applicable under small deviations from this architecture, like the ones mentioned above.

Errors can occur in any of the input processing stages described above. In Section 2.2 we analyze in more detail how understanding-errors come into being. For now, we focus our attention on the final product of the input processing stage, i.e. the discourse-level interpretation, illustrated in the grayed box in Figure 6. By definition, we say that an **understanding-error** occurs when **the discourse-level interpretation constructed by the system does not match the user's expressed intent**. This can happen in two ways: (1) the system constructs an incorrect discourse-level interpretation of the user's turn, or (2) the system fails altogether to construct a discourse-level interpretation. These two failure modes correspond to the two main types of understanding-errors affecting spoken language interfaces: **misunderstandings** and **non-understandings**.

If the system's discourse-level interpretation matches the user's expressed intent, we say that we have a **correct understanding**. Note that correct understanding can happen even in the presence of some input processing errors. For instance in the example given in Figure 6, the speech recognition component incorrectly misrecognized *"two p.m."* instead of *"until two p.m."*; nevertheless the final, discourse-level interpretation constructed by the system, `[end_time=2pm]`, matches the user's expressed intent.

2.1.2 Misunderstandings

By definition, we say that a **misunderstanding-error** (or **misunderstanding** in short) happens when **the system constructs an incorrect discourse-level interpretation of the user's turn**. Figure 7 shows two examples. In the first example, the recognition engine misrecognized *"Tuesday"* instead of *"Thursday"*. The recognition hypothesis is still semantically coherent. The language understanding module creates a corresponding semantic representation and forwards it to the dialog manager. The dialog manager integrates this incorrect information into the discourse structure, and continues the dialog. The system is not aware that a misunderstanding has occurred. In this first example, neither is the user. Several turns can pass until the user will finally notice that the system misunderstood. The user detects the misunderstanding only when the system echoes back the incorrect information (e.g. in turn 5 in our example), or presents some other information from which the user can infer that the system had misunderstood. In some cases, a misunderstanding might go unnoticed throughout the whole conversation.

In the second example, the user response, *"guest user"* is misrecognized as *"next Tuesday"*. Again, a coherent semantic interpretation is formed and the system believes that the user wants to change the date for the reservation to next Tuesday. Next, the system acts based on this incorrect information, and presents a new list of rooms to the user. In this case the user is able to detect immediately that the system misunderstood because the system immediately echoed back the information it thought it heard (turn 3.)

To guard against misunderstandings, spoken dialog systems typically rely on recognition confidence scores. Besides the recognition hypothesis, most speech recognizers also output a score (generally between 0 and 1), that reflects the estimated accuracy of the recognition result. This confidence score can be used by a spoken language interface to make error handling decisions. For instance, if the system is not very confident about a particular result, it might decide to ask the user to confirm that information, or even ignore (reject) it altogether. Note that confidence scores do not perfectly

Example 1: (misunderstanding in turn 2)

1 S: Welcome to RoomLine, the automated conference room reservation system. How may I help you?
2 U: *I need a reservation for next Thursday*
R: I NEED A RESERVATION FOR NEXT TUESDAY / 0.68
P: [date=next Tuesday]
3 S: Okay, for what time do you need the room?
4 U: *two to four p.m.*
R: TWO TO FOUR P.M. / 0.88
P: [start_time=2pm, end_time=4pm]
5 S: Let's see ... I found four room available Tuesday, June 27th between two and four p.m. Would you like a small room or a large one?

Example 2: (misunderstanding in turn 2)

1 S: Could I have your name please? If you are not registered with the system, just say 'guest user'.
2 U: *guest user*
R: NEXT TUESDAY / 0.51
P: [date=next Tuesday]
3 S: a room for Tuesday, June 27th ... I found three rooms available for Tuesday, June 27th between 2 and 4 p.m. Would you like a small room or a large one?

Figure 7. Example misunderstandings in a conference room reservation system
(S: marks the system turns, U: marks the user turns, R: marks the recognition result, P: marks the semantic representation of the recognition result)

reflect the system's recognition performance, as our examples illustrate. If the confidence scores were perfect, no misunderstandings would ever happen. Specifically, misrecognitions might happen, but the dialog manager would be aware of them and would avoid integrating incorrect information into the discourse. In Chapter 5 we discuss several methods for constructing accurate confidence scores. For now, it suffices to say that confidence scores are available in most systems, they are more or less accurate, and that they can be used to guard against misunderstandings.

2.1.3 Non-understandings

We now turn our attention to non-understandings. By definition, we say that a **non-understanding-error** (or **non-understanding** in short) happens when **the system fails to construct a discourse-level interpretation for the user's turn**[6]. Figure 8 provides a number of examples. For instance, in the first example, the user response *"two"* was incorrectly recognized as *"do you"*. The language understanding module was not able to construct a semantic interpretation for this recognition hypothesis. As a result, an empty semantic frame was forwarded to the dialog manager. The dialog manager was aware that the user said something, but did not obtain any semantic representation for this turn. As a result, the dialog manager was not able to form a meaningful interpretation of the user's turn at the discourse level. We call this a **no-input** non-understanding, since the dialog manager does not receive a semantic interpretation for the user's utterance.

Depending on the amount and the type of information that the dialog manager receives from the language understanding component, two other types of non-understandings can be defined: **unexpected-input** non-understandings, and **rejection** non-understandings.

An **unexpected-input** non-understanding occurs when the language understanding component generates a semantic representation, but the dialog manager cannot incorporate it into the current discourse structure. The second example in Figure 8 shows an unexpected-input non-understanding. The user response *"large"* was incorrectly recognized as *"March"*. The language understanding component constructed a corresponding semantic representation `[month=March]`, and forwarded it to the dialog manager. However, the dialog manager did not expect to hear a date at this point in the dialog, and therefore could not incorporate this answer in the discourse structure. Like in the first case, no discourse level interpretation was generated for the user's turn.

The third type of non-understanding, **rejection** non-understanding, occurs when the system deliberately rejects an input because of a low confidence score. Take for instance the third example in Figure 8. Although the language understanding component generated a semantic input that could be integrated in the current context, the system decided to reject it because of a low recognition confidence score. In effect, a non-understanding error was deliberately created by the system, in order to avoid a potential misunderstanding. Note however that a low confidence score (as in example 3) does not necessarily mean that the recognition result (or the constructed semantic representation) was incorrect – confidence scores do not always reflect the accuracy of the system understanding. As a result, sometimes the system might incorrectly reject an accurate semantic input, such as in the example above. In this case, we say we have an incorrect- or **false-rejection**. In contrast, if the system rejects an inaccurate semantic representation, then we have a correct- or **true-rejection**.

In the sequel, we will use the term **genuine non-understanding** to denote no-input and unexpected input non-understandings, and distinguish them from rejection non-understandings. In the first two cases, the system is not able to form a meaningful discourse-level interpretation of the user's utterance. In the last case, the system is able to form an interpretation, but actively chooses not to; the system creates a non-understanding in order to avoid a potential misunderstanding.

---

[6] Note that our term **non-**understanding is different from **not-**understanding, as used by McRoy and Hirst in [73]. In their work, McRoy and Hirst define not-understanding as "a participant's failure to obtain any complete and unique interpretation of an utterance". A not-understanding therefore can happen if the participant cannot obtain an interpretation (same as our non-understanding), but also if a participant obtains an interpretation that is either incomplete or ambiguous.

Example 1: (no-input non-understanding in turn 2)

1 S: Until what time will you need this conference room?
2 U: *two*
R: DO YOU / 0.33
P: []

Example 2: (unexpected-input non-understanding in turn 2)

1 S: I found four rooms available in Wean Hall. Would you like a small room or a large one?
2 U: *large*
R: MARCH / 0.45
P: [month=March]

Example 3: (rejection non-understanding in turn 2)

1 S: How else can I help you today?
2 U: *I need to make sure the room will hold forty people and has a network connection and a data projector*
R: I NEED TO RESERVE A ROOM FOR HOLD FORTY PEOPLE AND HAS A NETWORK CONNECTION BENNETT DATA PROJECTOR / 0.14
P: [size=40, equipment=network;projector]

Figure 8. Example non-understandings in a conference room reservation system (S: marks the system turns, U: marks the user turns, R: marks the recognition result, P: marks the semantic representation of the recognition result)

2.1.4 The misunderstandings / non-understandings trade-off

Whenever used, the rejection mechanism described above introduces a trade-off between misunderstandings and non-understandings. Generally, systems decide whether or not to reject an utterance by comparing the confidence score against a preset rejection threshold. By changing this threshold, the system can change the relative proportions of misunderstandings and non-understandings that occur in the conversation. Figure 9 illustrates this trade-off. As the rejection threshold increases from 0 (no rejections) to 1 (all utterances are rejected) the misunderstanding error rate (i.e. the proportion of misunderstandings in a conversation) decreases from a maximum value to 0%. At the same time the non-understanding error rate increases from an initial non-zero value (due to no-input and unexpected-input non-understandings) to 100%. Later on, in Chapter 7, we discuss this trade-off in more detail and present a data-driven approach for optimizing dialog-state specific rejection thresholds with respect to a chosen global performance metric.

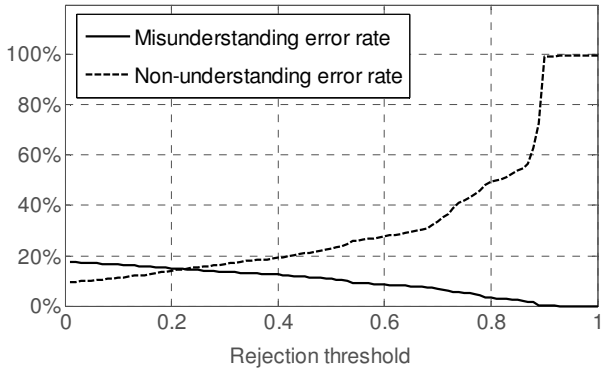


Figure 9. Trade-off between misunderstandings and non-understandings (as a function of rejection threshold) in a conference room reservation system

2.2 Sources of understanding-errors

In the previous section, we have defined precisely two types of understanding-errors commonly encountered in spoken language interfaces: non-understandings and misunderstandings. We now turn our attention to the main sources of these errors. We describe a four-level model for error-source analysis and empirically analyze the distribution of error sources in two deployed spoken dialog systems.

2.2.1 A four-level model for error source analysis

As a starting point for the error source analysis, we used Clark's model of grounding in human-human communication [27]. In Clark's model, participants in a conversation (let's call them A and B) collaborate and coordinate on four different levels to establish mutual ground. At the first level, the "pre-linguistic, or nonlinguistic level" (channel) A executes a behavior t, and B attends to it. At the second level (or the signal level) A conveys a linguistic signal s to B and B identifies this signal. At the third level (intention), A is signaling a certain intention p to B, and B is understanding A's intention. Finally, at the last level (conversation), A is proposing a collaborative activity w to B, and B is considering A's proposal.

Clark's original grounding model has already been used as a basis for error classification in the context of human-machine interaction. In [81], Paek uses Clark's model as the starting point for constructing a unified taxonomy of communication errors. Paek argues that various classifications of communication errors have been proposed in different disciplines (e.g. conversation analysis, second language acquisition, computational linguistics, etc), but that little effort has been made to compare and contrast those classifications. Paek's proposed taxonomy is anchored in Clark's model and aims to bridge insights from different areas of research into an interdisciplinary approach. In [109], Schlangen also uses Clark's model as a basis for identifying possible causes for requesting clarifications in dialog. Schlangen argues that the distinctions made by Clark's model are still fairly coarse-grained and he suggests a further decomposition of the third level.

In this work, we use Clark's model and Paek's taxonomy as starting points. In addition, we make the observation that a mapping (or correspondence) can be established between: (1) the four levels of grounding in Clark's model, (2) the flow of information from the user to the system, and (3) the various components in the system's input processing line. This correspondence is illustrated in Figure 10. At the conversation level, the user has a high-level goal (w). Subsequently this goal acquires a corresponding semantic (p), lexical (s) and eventually an acoustic (t) representation in the lower levels. The acoustic signal then passes through a noisy channel, and arrives at the system side. Here, a

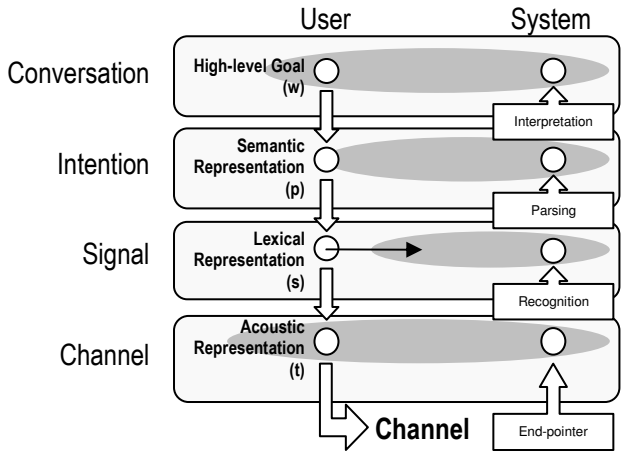


Figure 10. A view of grounding in human-computer communication

series of chained components – end-pointer, speech recognition, language understanding, and discourse interpretation – are used to progressively reconstruct the user's higher level goal from the incoming acoustic signal.

Understanding-errors typically occur when a mismatch happens (on a certain level) between the expressed form of the user's intent and the system's modeling and/or recognition abilities. For example, at the conversation level, the user might not be aware of the system's scope and limitations and might start off with a goal which the system cannot handle. For instance the user might ask if a certain room has windows in a conference room reservation system. In this situation, because the system does not model this type of query, it will be impossible to correctly reconstruct the user's goal (`w`); an understanding-error will be inevitable. Similarly, at the signal level, mismatches between a user's pronunciation style and the system's acoustic models can lead to speech recognition errors, and ultimately to understanding-errors.

This view of understanding-errors highlights two complementary approaches that can be used to mitigate the mismatches. The first approach is to create models that provide better coverage, while still maintaining good performance (i.e. enlarge the ovals in Figure 10). For instance, at the signal level, we could expand the system's language (and acoustic) models to capture more variability in the user's speech. The second approach is to steer the user's responses into the space covered by the system's models. For instance, carefully designed "you-can-say" help prompts can inform users about the best (in-language-model) way to express a request. Both approaches pose their own sets of challenges. Enlarging the system's models raises difficulties in preserving system performance since recognition is a search problem: a larger search space in general leads to more difficult problem. Shaping the user's responses is not an easy task either. Finding the best ways for entraining the users, without increasing frustration is an active research area [123]. In general, this type of approach can be used to address "language domain" problems, such as ungrammatical [123], out-of-domain, or out-of-application scope utterances. Acoustic and channel-level problems are harder to address in this framework (how can we entrain someone to speak without an accent?)

Coming back to the error source analysis, we identify four major sources of errors, based on the level at which the mismatch occurs:

- **Out-of-Application (OOA)** [Conversation Level]: The user's utterance falls outside the application's functionality. These errors can be further divided into **out-of-domain** utterances (e.g. the user asks the room-reservation system about the weather), and **out-of-application-scope** utterances, i.e. utterances that express in-domain goals but which the system is however not able to handle (e.g. the user asking for an uncovered bus route in a bus information system.)
- **Out-of-Grammar (OOG)** [Intention Level]: The user's utterance is within the domain and scope of the application, but outside of the system's semantic grammar (e.g. the user says *"erase reservation"*, which is not in the system's grammar; the system could have handled the request had the user said *"cancel reservation"* or *"delete reservation"*, which are in the system's grammar.)
- **ASR Error (ASR)** [Signal Level]: The user's utterance is within the application's domain, scope and grammar, but is not recognized correctly due to acoustic or statistical language modeling mismatches (e.g. the user says *"Thursday morning"* but this is misrecognized as *"`Friday morning`"*.)
- **End-pointer Error** (END) [Channel Level]: The end-pointer is not able to correctly segment the incoming audio signal (e.g. it truncates the utterance or sends an empty utterance into the input line.)

We used this taxonomy to label the sources of misunderstandings and non-understandings in two different spoken dialog systems: RoomLine and the Let's Go! Public Bus Information System. The empirical results are discussed below.

2.2.1.1 Empirical error source analysis in the RoomLine domain

We first analyzed the sources of misunderstandings and non-understandings in a corpus collected with RoomLine, a deployed spoken dialog system that provides access to conference room schedule information and allows users to make conference room reservations. The system has access to live schedules for 13 conference rooms in 2 buildings on the CMU campus, and to the various characteristics of these rooms such as location, size, network access, whiteboards, and audio-visual equipment. To perform a room reservation, the system finds the list of rooms that satisfy an initial set of user-specified constraints. Next, RoomLine presents this information to the user, and engages in a follow-up negotiation dialog to identify which room best matches the user's needs. Once the desired room is identified, registered users can authenticate using a 4-digit touch-tone PIN, and the system performs the reservation through the campus-wide CorporateTime calendar server. More details about this system are presented later, in Chapter 3, subsection 3.4.1.

The corpus used for the error source analysis was collected during a user study with this system, described in detail in Chapter 8, subsection 8.3.1. Here, it suffices to say that the experiment was a controlled user study in which first-time users performed up to 10 scenario-based interactions with the system. We used for our analysis a portion of this corpus (collected in the `control` condition of the above-mentioned experiment) containing 226 dialogs and 4012 user utterances. Each misunderstanding and non-understanding in the corpus was manually annotated with a tag indicating the error source, according to the classification scheme described we have previously described.

The breakdown of misunderstandings and non-understandings by error source is shown in Table 2 and illustrated in Figure 11. Most errors (62% of non-understandings and 84% of misunderstandings) originate at the signal or speech recognition level. At the same time, a large number of non-understandings, and a smaller but still significant number of misunderstandings are caused by out-of-application and out-of-grammar utterances, as well as end-pointing errors.

The out-of-application errors encountered in the RoomLine data consist almost entirely of out-of-application-scope utterances. These utterances are in-domain, but they refer to inexistent application functionality (We believe that the lack of out-of-domain utterances is most likely due to the scenario-driven nature of the interactions; participants in the experiment were given a brief explanation about the system prior to the interaction.) A closer inspection of the out-of-application-scope utterances revealed that they subsume roughly equal numbers of requests for inexistent task-level functionality (e.g. *"I need a room for Monday or Tuesday"* – the system does not handle "or" requests), and requests for inexistent meta functionality, such as *"go back!"* or various types of corrections (e.g. *"You got the wrong day!"*, *"Change the date!"*, *"The time is wrong"*, etc.)

Together with the out-of-grammar utterances, the out-of-application utterances reflect one facet of an existing mismatch between user and system at the intention and conversation levels. A

	Non-understandings	Mis-understandings
OOA	16.8%	6.7%
OOG	15.3%	6.1%
ASR	62.2%	84.5%
END	5.7%	2.7%
TOTAL	100%	100%

Table 2. Error sources for non-understandings and misunderstandings in the RoomLine system

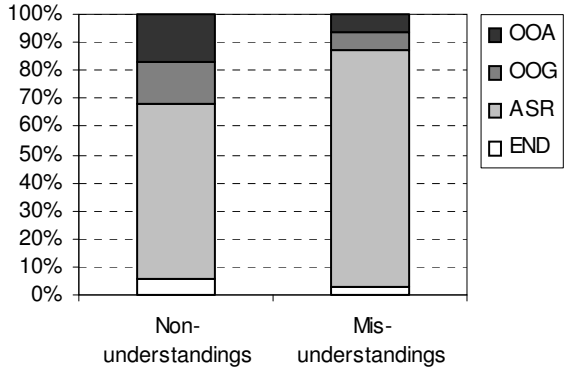


Figure 11. Error sources for non-understandings and misunderstandings in the RoomLine system

second interesting facet, revealed through an analysis of the transcripts, is that there are certain aspects of system functionality which are never (or very rarely) addressed by the users. For instance, although the users were told during the pre-experiment briefing that they can say *"Help"* to the system at any point in time, this function was invoked in only in 7 of 226 sessions. Other types of help commands like *"where are we?"*, *"what can you do?"*, *"what can I say?"*, *"interaction tips"*, although available at all times, were not discovered by the users and therefore were never used. We found similar examples with respect to task-level functionality for commands like *"tell me all the rooms"*, *"I want a smaller / larger room"*, *"I don't care"* (about room size), *"how big is this room"*, *"tell me about this room"*, etc. Based on these observations, we conjecture that, apart from out-of-grammar errors, users are also not aware of the full functionality of the application. The fairly large number of out-of-application and out-of-grammar utterances suggests that the number of non-understandings could be reduced by better informing the users about the application capabilities and boundaries as well as directing them into this space. How exactly this shaping can be performed remains an active research issue [123].

The majority of non-understandings – 62% (and even more so for misunderstandings – 84%) originate at the speech recognition level. Although a large number of contributing factors can be identified, precise blame assignment is harder to perform. For instance, non-native accents have a significant impact on ASR performance: average word-error-rate was 20.7% for natives, versus 42.3% for non-natives in this experiment. Ambient noises also have a pronounced effect on recognition performance: average word-error-rate for noisy utterances was 32.8%, significantly larger than 25.1% for noise-free utterances (the percentages were computed based on noise labels present in the ortographic transcription of the user utterances.) Other factors, such as speaking rate, user frustration, hyper-articulation, have also been shown to correlate well with recognition accuracy [25].

Finally, a small proportion of the non-understandings (6%) and misunderstandings (3%) in the RoomLine corpus were caused by end-pointing errors. In most of these cases the end-pointer prematurely (and therefore falsely) detected an end-of-utterance, chopping the user's turn into two parts and leading to an understanding-error.

In Figure 12 and Table 3 we also show how often each of the four error sources led to misunderstandings and non-understandings. Out-of-application, out-of-grammar and end-pointing errors lead in roughly equal amounts to both misunderstandings and non-understandings. At the same time, three out of four (more precisely 72%) of the signal-level (or speech recognition) errors lead to misunderstandings, while only one out of four results in a non-understanding.

	OOA	OOG	ASR	END
Misunderstandings	42.4%	42.6%	71.6%	46.5%
Non-understandings	57.6%	57.4%	28.4%	53.5%
TOTAL	100.0%	100.0%	100.0%	100.0%

Table 3. Percentages of utterances within each error source that lead to misunderstandings and non-understandings

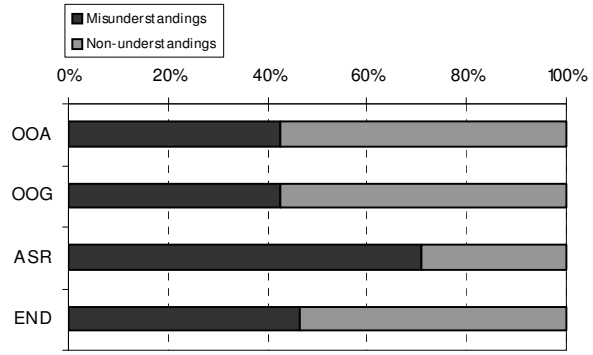


Figure 12. Percentages of utterances within each error source that lead to misunderstandings and non-understandings

§ Rejection non-understandings

So far the discussion has focused on **genuine non-understandings**, i.e. situations in which the input is corrupted to the point that the system is no longer able to construct a meaningful discourse-level interpretation of the user's turn. However, as we have seen in subsection 2.1.3, the dialog manager also uses rejections to guard against potential misunderstandings: if the system obtains an interpretation of the user's input, but the confidence score is below a preset threshold, then the utterance will be rejected. Figure 13 illustrates the ratios of non-understandings and misunder-standings, as computed before and after the rejection mechanism. After rejections, the total ratio of non-understandings grows by 7.1% absolute from 10.1% to 17.2%. About 40% of the rejections (2.9% of the total number of turns, and 17% of the total number of non-understandings) are false-rejections, i.e. utterances correctly understood but falsely rejected because of a low confidence score. The relatively high false rejection rate contributes significantly to the total number of non-understandings, which is on par with other sources of errors. The false-rejection rate can be lowered by building better confidence annotators, or by tuning the rejection threshold to the domain.

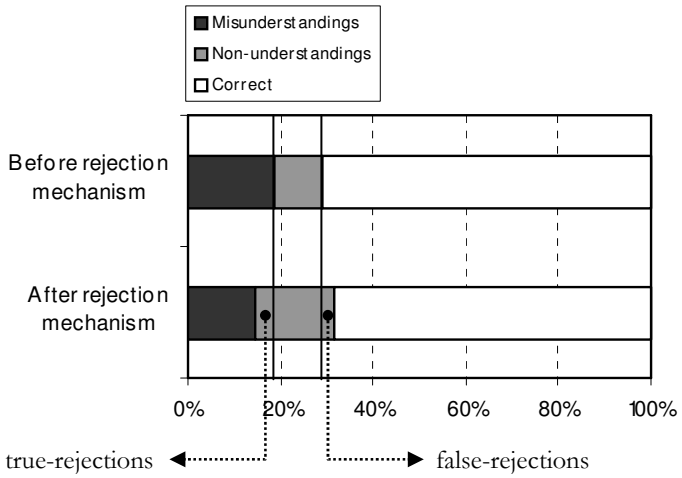


Figure 13. Misunderstandings and non-understandings before and after rejection in the RoomLine system

Overall, the current limitations in the speech recognition technology seem to be the most important source of misunderstandings and non-understandings. Out-of-application and out-of-domain utterances were a second important source of errors, followed by end-pointer errors and false-rejections.

2.2.1.2 Empirical error source analysis in the Let's Go! Public domain

In an effort to create a more comprehensive picture and to understand how well the observations we made above generalize to other domains, we conducted a second analysis based on data from another deployed spoken dialog system: the Let's Go! Public Bus Information system [88, 89]. Let's Go! public is a telephone-based spoken dialog system that provides access to bus route and schedule information. The system knows about 12 bus routes, and 1800 place names in the greater Pittsburgh area. In order to provide bus schedule information, the system tries to identify the user's departure and arrival stop, and the departure or arrival time. Once the results are provided, the user can ask for the next or previous bus on that route, or can restart the conversation from the beginning to get information for a different route. In contrast to the RoomLine system, Let's Go! Public has a system-initiative interaction style.

The system is connected live to the Port Authority of Pittsburgh customer service line during non-business hours; the corpus we analyze here was collected during one of the first weeks of deployment, and consists of 186 calls (2968 utterances) from users with real needs (in contrast to the

A. Let's Go! Public

	Non-understandings	Mis-understandings
OOA	14.4%	10.7%
OOG	11.3%	6.3%
ASR	55.0%	73.5%
END	19.3%	9.5%
TOTAL	100.0%	100.0%

B. RoomLine

	Non-understandings	Mis-understandings
OOA	16.8%	6.7%
OOG	15.3%	6.1%
ASR	62.2%	84.5%
END	5.7%	2.7%
TOTAL	100.0%	100.0%

Table 4. Error sources for non-understandings and misunderstandings in the Let's Go! Public and RoomLine systems

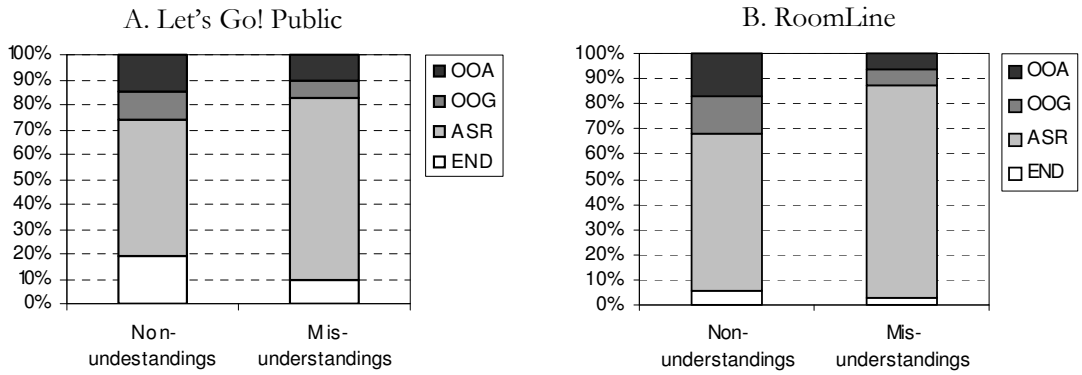


Figure 14. Error sources for non-understandings and misunderstandings in the Let's Go! Public and RoomLine systems

scenario driven interactions from the RoomLine domain.)

The breakdown of misunderstandings and non-understandings by error source in the Let's Go! Public data is shown in Table 4.A and Figure 14.A. For comparison purposes, we show again the same results for the RoomLine data in Table 4.B, Figure 14.B. The breakdown of misunderstandings and non-understandings in the four different error sources is similar across the two systems. In both cases, the largest proportion of errors is caused by signal-level (i.e. speech recognition) errors. In both cases, this proportion is larger in the case of misunderstandings. Out-of-application and out-of-grammar errors contribute in about equal proportions to the overall number of misunderstandings and non-understandings in both domains.

Despite the similarity in the proportion of out-of-application utterances, the breakdown of these errors into out-of-application-scope and out-of-domain utterances is different. In the previous section, we have seen that in the RoomLine data most of the out-of-application errors were out-of-application scope. The lack of out-of-domain utterances in the RoomLine data was explained by the scenario driven nature of the interactions and the controlled experiment aspects of the data collection process. In contrast, in the Let's Go! Public data the largest number (65%) of out-of-application utterances are out-of-domain utterances. They generally contain various remarks and expressions of frustration, or user speech that is not directed to the system. The out-of-application-scope utterances mostly contain bus routes or bus stop addresses that are not covered by the system.

The most striking difference between the two domains regards the proportion of understanding-errors caused by end-pointing errors (see Figure 14 and Table 4.) The numbers are significantly larger in the Let's Go! Public domain: 19.3% versus 5.7% of the non-understandings and 9.5% versus 2.7% of misunderstandings. We believe the difference is explained by the more adverse environmental conditions in which the Let's Go! Public system operates. The system receives calls at night (during non-business hours) from users with real-needs. Oftentimes, the calls are made from public phones or cellular phones on the street. As a consequence, there is a much larger variability in this corpus in terms of environmental noise, quality of input channel, as well as speaking style. Anec-

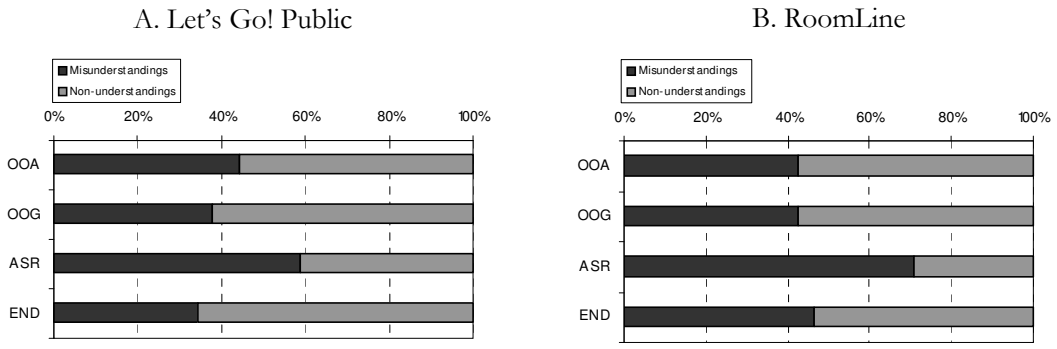


Figure 15. Percentages of utterances within each error source that lead to misunderstandings and non-understandings in the Let's Go! Public and RoomLine systems

dotally, we have observed a large number of turn-overtaking problems, i.e. situations in which the user and the system barge-in on each other. Overall, end-pointing is a more challenging problem in the Let's Go! Public domain.

We again analyzed how often each error source leads to misunderstandings and non-understandings. The results for the two domains are shown in Figure 15. The pattern that emerges is similar across the two domains: signal-level errors lead more often to misunderstandings than to non-understandings; in comparison, out-of-application, out-of-grammar and end-pointer errors lead to roughly equal amounts of misunderstandings and non-understandings.

In summary, a number of similarities in the distribution of various sources of errors was found across the two domains. In both datasets, the largest proportion of understanding-errors is caused by speech recognition (i.e. signal-level) errors. This proportion is larger in the case of misunderstandings (conversely, signal-level errors lead more often to misunderstandings than to non-understandings). Language domain-errors, such as out-of-application and out-of-grammar utterances, contribute in a smaller but still significant proportion to the overall number of errors. At the same time, domain-specific differences can also be identified. For instance, we have seen a significantly larger number of end-pointing and out-of-domain errors in the Let's Go! Public system, caused ultimately by a more challenging operating environment.

The empirical analysis of error sources described in this section confirms the anecdotal observation that the speech recognition layer constitutes one of the main sources of errors in spoken language interfaces. At the same time, the differences observed across the two datasets indicate that a system-specific analysis of error sources can more accurately point to certain trouble spots, and help focus development and tuning efforts (e.g. a better end-pointing algorithm would probably lead to significant performance increases in the Let's Go! Public domain.)

2.3 Impact of understanding-errors on dialog performance

In the previous two sections in this chapter we have introduced two types of understanding-errors that commonly affect spoken dialog systems (i.e. misunderstandings and non-understandings), and investigated the main sources of these errors. We now turn our attention to a second important question: what is the impact of these understanding-errors on overall dialog performance?

Understanding-errors clearly obstruct the normal progress of a conversation. During non-understandings, the system fails to construct a discourse-level interpretation of the user's turn. No useful information is acquired by the system, and no progress is made. During misunderstandings, the system acquires incorrect information, which will have to be corrected later. In general, understanding-errors lead to longer dialogs and increased user frustration. When present in large numbers, they lead to total communication breakdowns. Intuitively, these errors clearly exert a significant negative impact on system performance. However, we would like to be able to better qualify and quantify this effect.

In this section, we present a data-driven method for quantitatively assessing the impact of understanding-errors on global dialog performance. We show that misunderstandings and non-understandings have different dialog costs, and that these costs vary across domains and systems. The proposed analysis technique allows us to infer the costs from data, and provides a detailed view of the impact of understanding-errors on performance.

2.3.1 A data-driven approach for error-cost analysis

We introduce the proposed approach with an example. Suppose we are interested in assessing the impact of misunderstandings and non-understandings on the probability of task success[7]. To perform this assessment we construct a regression model that relates the ratio of misunderstandings and non-understandings in the dialog to overall task success. Each data-point in the model corresponds to a dialog session. The independent variables will be the misunderstanding error rate (%MIS in the sequel) and the non-understanding error rate (%NONU in the sequel). These variables are computed for each session. The dependent variable in the regression model is a task success indicator (TS).

$$TS \leftarrow \%MIS + \%NON$$

The type of regression used should be adapted to the underlying distribution for the dependent variable. In this case, since task success is a binary measure (0 marks a failed dialog, 1 marks a successful dialog), a logistic regression model is appropriate. The model is:

$$\log\left(\frac{P(TS=1)}{P(TS=0)}\right) = \alpha + \beta \cdot \%MIS + \gamma \cdot \%NON$$

or

$$P(TS=1) = \frac{e^{\alpha+\beta\cdot\%MIS+\gamma\cdot\%NON}}{1+e^{\alpha+\beta\cdot\%MIS+\gamma\cdot\%NON}}$$

Alternatively, if we were interested in studying the effect on task duration (TD), measured as turns to completion, a Poisson model would be more appropriate:

$$\log(TD) = \alpha + \beta \cdot \%MIS + \gamma \cdot \%NON$$

Once the model is fit, the regression coefficients characterize the impact of misunderstandings and non-understandings on performance. Assume for instance that we find the following fit (this in fact is the fit obtained in the RoomLine domain, and described in more detail in the next subsection):

$$\log\left(\frac{P(TS=1)}{P(TS=0)}\right) = 3.79 - 11.28 \cdot \%MIS - 4.36 \cdot \%NON$$

The regression coefficients indicate that both misunderstandings and non-understandings exert a significant effect on performance, and that misunderstandings are about three times more costly than non-understandings in this domain (i.e. -11.28 versus -4.36).

The proposed analysis technique bears similarities to the PARADISE evaluation framework [127, 128]. In both cases, multivariate regression models are used to model relationships between various factors and overall dialog performance. PARADISE posits user satisfaction as the overall evaluation criterion, and uses two types of factors to predict user satisfaction: on one hand task success, and on the other hand a battery of other factor modeling interaction costs (e.g. word-error-rate, barge-ins, etc.) In our proposed methodology, we focus on identifying the relationship between un-

[7] Depending on the domain, different global dialog performance metrics might be of interest. For instance, in a tutoring spoken language interface learning gains would be a more appropriate metric. The error cost analysis method presented in this section can be applied with respect to any global performance metric.

derstanding-errors and global objective performance metrics such as task success and task duration. The relationship to subjective dialog performance metrics could also be investigated in the same manner. The analysis would be however more difficult from a practical standpoint: subjective metrics, such as survey-elicited user-satisfaction scores, can be strongly influenced by a number of other random factors, such as user expectations and previous experiences using language technology. In the absence of good normalization techniques [48], or large amounts of data, this increases the difficulty of teasing appart the effects of various understanding-errors.

Next, we discuss empirical results obtained by applying the proposed approach in two spoken dialog systems: RoomLine and the Let's Go! Public Bus Information System.

2.3.1.1 Experimental results in the RoomLine system

We first used the proposed approach to assess the impact of non-understandings and misunderstandings on global dialog performance in the RoomLine domain. The analysis was based on data collected in the same user study we mentioned above. The collected corpus contained 449 dialogs. We eliminated sessions with less than 3 turns and sessions with differences between perceived and objective task completion. The final corpus contained 411 dialogs. The average task success rate in this corpus was 81.27%.

We began by investigating the impact of misunderstandings on the probability of task success. The independent variable is the misunderstanding error rate (%MIS). The dependent variable is the binary task success (TS). Table 5 shows the result of the fit. The model confirms that misunderstandings exert a significant negative impact on the probability of task success: the regression coefficient for %MIS is negative, -12.13, and statistically significant, $p<10^{-4}$. On the training set, the fitted model increases the average log-likelihood of the data from -0.4824 to -0.3592. A similar result is obtained when using a 10-fold cross-validation process, indicating a robust fit. In Figure 16 we show the predicted probability of task success P(TS=1), as a function of the misunderstanding error rate %MIS, together with 95% confidence bands. The dots at P(TS=1)=100% represent successful dialog sessions; the dots at P(TS=1)=0% represent failed dialog sessions. The star marks the average misunderstanding error rate and the average task success rate in the dataset. A histogram of the per session misunderstanding error rate is also shown. As the misunderstanding error rate increases, the probability of task success drops sharply. As Figure 16 illustrates, the dependence is not linear. For instance, a 10% reduction of the misunderstanding error rate from 30% to 20% corresponds to a 29% increase in the probability of task success, from 39% to 68%. However, a 10% reduction in misunderstanding error rate from 15% to 5% corresponds to only a 13% increase in the probability of task success, from 80% to 93%. The proposed model provides a detailed quantitative assessment of the effect of misunderstandings on task success. This type of information can be very useful in channeling system optimization and tuning efforts.

Next, we built a similar model for assessing the impact of the non-understanding error rate (%NONU) on the probability of task success (TS). The model is shown in Table 6, and illustrated in Figure 17. Non-understandings also exert a significant negative effect on task success even though the effect is less pronounced (the drop in the curve is less sharp). Note that the estimated probability of task success is less reliable when the non-understanding error rate is above 35-40% - the confidence bounds are wider due to data sparsity issues. Our corpus contains few sessions with a very high non-understanding error rate, and some of these sessions happen to actually be successful.

Next, we constructed a third model that takes into account both misunderstandings and non-understandings. The results are presented in Table 7. Both the misunderstanding error rate (%MIS) and non-understanding error rate (%NONU) have a significant negative impact on the probability of task success. The model provides a precise quantitative assessment of this impact. An analysis of the regression coefficients reveals that misunderstandings are almost three times more costly (-11.27) than non-understandings (-4.35). This finding corresponds to the intuition that acquiring false information is more detrimental than not acquiring any information. The effect is also visible in Figure 18, which shows the estimated probability of task success at different misunderstanding

logit(TS) = 3.19 – 12.14 · %MIS			
	Coef.	p-value	S.E.
(constant)	3.1900	$< 10^{-4}$	0.2792
%MIS	-12.1375	$< 10^{-4}$	1.4438

Model	AVG-LLIK	HARD
Baseline	-0.4824	18.73%
Model (train)	-0.3592	14.36%
Model (CV)	-0.3680	14.37%

Table 5. Model for impact of misunderstandings on task success in the RoomLine system

logit(TS) = 2.51 – 6.29 · %NONU			
	Coef.	p-value	S.E.
(constant)	2.5110	< 10-4	0.2476
%NONU	-6.2919	< 10-4	1.1133

Model	AVG-LLIK	HARD
Baseline	-0.4824	18.73%
Model (train)	-0.4405	18.25%
Model (CV)	-0.4466	18.50%

Table 6. Model for impact of non-understandings on task success in the RoomLine system

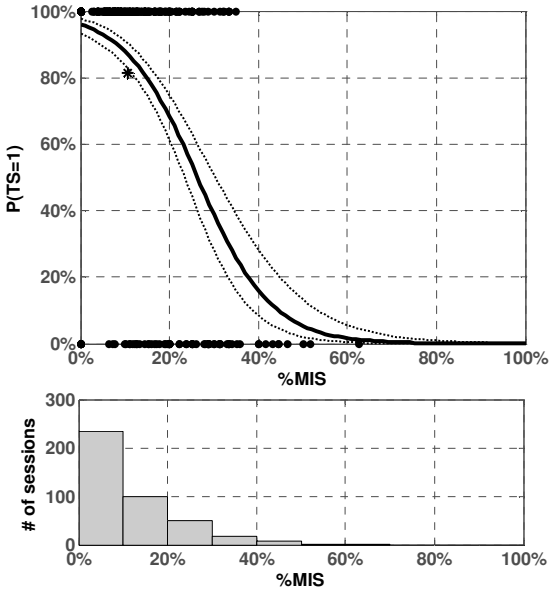


Figure 16. Probability of task success as a function of the misunderstanding error rate (%MIS) in the RoomLine system

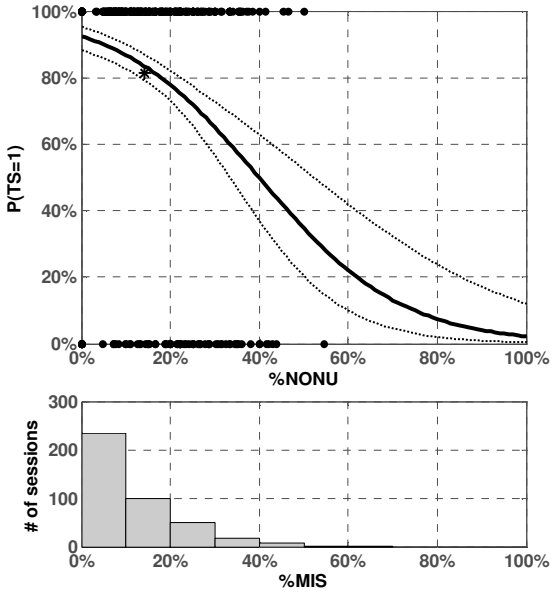


Figure 17. Probability of task success as a function of the non-understanding error rate (%NONU) in the RoomLine system

and non-understanding error rates.

Lastly, we constructed a model that also included the average per session word-error-rate (WER) as a dependent variable in the model (besides %MIS and %NONU):

$$\log\left(\frac{P(TS=1)}{P(TS=0)}\right) = \alpha + \beta \cdot \%MIS + \gamma \cdot \%NON + \delta \cdot WER$$

This model did not produce an improved fit when compared to the model shown in Table 7. The resulting WER coefficient was not statistically significantly different than zero (p=0.2012). The model therefore indicates that the misunderstanding and non-understanding error rates already fully account for the impact of speech recognition errors on the system's performance.

Before concluding, it is important to note that the error-cost assessment models described above are correlational, rather than causative. In other words, the models cannot be used to draw firm conclusions like: "reducing the non-understanding error rate by x% will lead to an improvement of y% in task success". However, the models provide a quantitative assessment of the impact of understanding-errors on a chosen global performance metric. They can be used to make predictions which in turn can be useful in focusing optimization efforts. For instance, in the current RoomLine system, it seems that reducing the misunderstanding error rate is more important than reducing the non-understanding error rate. Such predictions remain however to be confirmed empirically.

logit(TS) = 3.79 – 11.28 · %MIS – 4.36 · %NONU			
	Coef.	p-value	S.E.
(constant)	3.7908	< 10-4	0.3539
%MIS	-11.2771	< 10-4	1.4755
%NONU	-4.3586	0.0006	1.2645

Model	AVG-LLIK	HARD
Baseline	-0.4824	18.73%
Model (train)	-0.3445	13.63%
Model (CV)	-0.3547	14.38%

Table 7. Model for impact of misunderstandings and non-understandings on task success in the RoomLine system

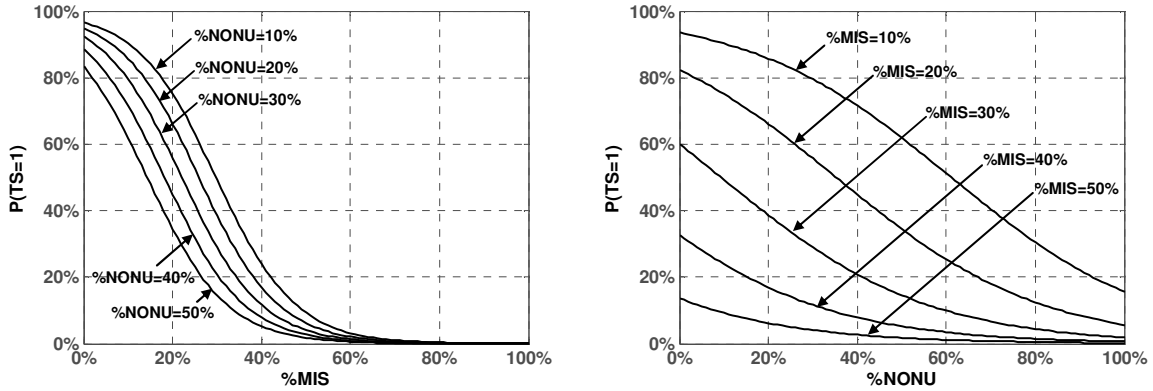


Figure 18. Probability of task success as a function of the misunderstanding (%MIS) and non-understanding (%NONU) error rates in the RoomLine system

2.3.1.2 Experimental results in the Let's Go! Public system

A second set of experiments were performed using data collected with the Let's Go! Public Bus Information system. The corpus contained 467 dialogs collected in the first month the system was released to the larger public (March 2005). Again, we eliminated sessions with less than 3 turns. The resulting corpus contains 432 dialogs. The average task success rate in this corpus is 56.25%. This lower average task success rate is to a large extent explained by the more difficult operating conditions for this system.

The model for assessing the impact of misunderstandings on the probability of task success is shown in Table 8 and illustrated in Figure 19. For comparison purposes, Figure 19 also shows the fitted probability of task success in the RoomLine system. As expected, misunderstandings also exert a significant negative impact on performance in the Let's Go! Public system. The impact is however different than in the RoomLine domain. When the misunderstanding error rate is in the range of 0-40%, the impact on performance is mostly linear. The estimated probability of task success is lower than in the RoomLine system in this range. Additionally, the Let's Go! Public system seems to be a bit more robust at high misunderstanding error rates (above 40%).

The model for assessing the impact of non-understandings is shown in Table 9 and illustrated in Figure 20. Here, the task success profile is more similar to the RoomLine system than in the case of misunderstandings.

The third model, shown in Table 10, provides an assessment of the combined effect of misunderstandings and non-understandings on task success. Again, both misunderstandings and non-understandings exert a significant negative impact on performance. The resulting regression coefficients show that the cost for misunderstandings (-8.17) is similar to the cost of non-understandings (-9.26) in the Let's Go! Public system. The relative costs are different when compared to the RoomLine system; in that domain misunderstandings were significantly more costly than non-

logit(TS) = 0.90 – 5.47 · %MIS			
	Coef.	p-value	S.E.
(constant)	0.8964	< 10-4	0.2066
%MIS	-5.4686	< 10-4	0.8936

Model	AVG-LLIK	HARD
Baseline	-0.6853	43.75%
Model (train)	-0.6328	36.81%
Model (CV)	-0.6365	36.77%

Table 8. Model for impact of misunderstandings on task success in the Let's Go Public system

logit(TS) = 1.87 – 6.71 · %NONU			
	Coef.	p-value	S.E.
(constant)	1.8703	< 10-4	0.2376
%NONU	-6.7063	< 10-4	0.7089

Model	AVG-LLIK	HARD
Baseline	-0.6853	43.75%
Model (train)	-0.5246	27.08%
Model (CV)	-0.5295	26.40%

Table 9. Model for impact of non-understandings on task success in the Let's Go Public system

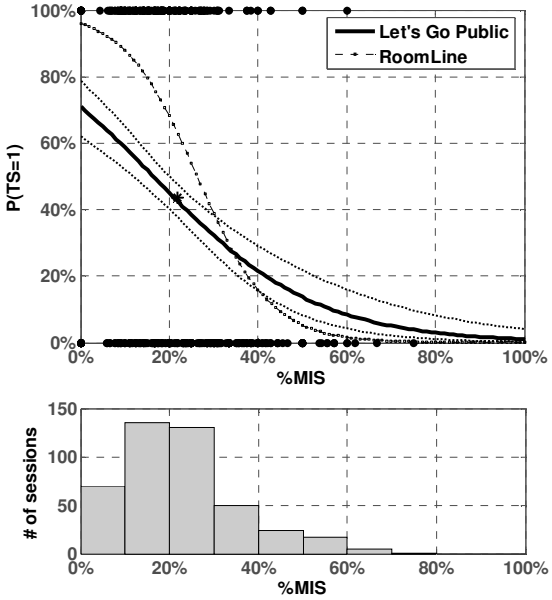


Figure 19. Probability of task success as a function of the misunderstanding error rate (%MIS) in the Let's Go Public and RoomLine systems

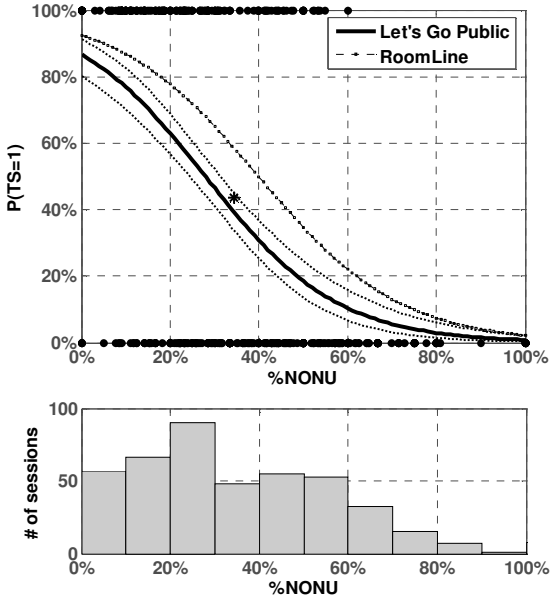


Figure 20. Probability of task success as a function of the non-understanding error rate (%NONU) in the Let's Go Public and RoomLine systems

understandings. A potential explanation lies in the confirmation policy used by the Let's Go! Public system. This system always explicitly confirmed each piece of information obtained from the users. This aggressive confirmation policy reduces the relative cost of misunderstandings. In contrast, since RoomLine does not always explicitly confirm, misunderstandings can go undetected (by both the system and the user) for a longer period of time. Corrections are harder to perform at a later stage, and as a result misunderstandings have a more pronounced overall impact in the RoomLine domain.

Again, adding the average session word-error-rate (WER) to the list of predictor variables did not further improve the fit for the model. The p-value for the WER regression coefficient was 0.9492, indicating again that the misunderstanding and non-understanding error rates fully characterize the impact of poor speech recognition on task success.

2.4 Summary

In this chapter, we closely investigated the two types of understanding-errors that commonly affect spoken language interfaces: misunderstandings and non-understandings. We started by providing precise definitions for these terms, identifying different types of non-understandings, and pointing to an inherent trade-off between these errors that appears in most interfaces. Then, we focused our attention on two questions: (1) what are the main sources of these understanding-errors, and (2) how large is their impact on overall dialog performance?

logit(TS) = 4.37 – 9.29 · %MIS – 8.17 · %NONU			
	Coef.	p-value	S.E.
(constant)	4.3736	< 10-4	0.4436
%MIS	-9.2856	< 10-4	1.2186
%NONU	-8.1680	< 10-4	0.8191

Model	AVG-LLIK	HARD
Baseline	-0.6853	43.75%
Model (train)	-0.4282	19.91%
Model (CV)	-0.4354	19.66%

Table 10. Model for impact of misunderstandings and non-understandings on task success in the Let's Go! Public system

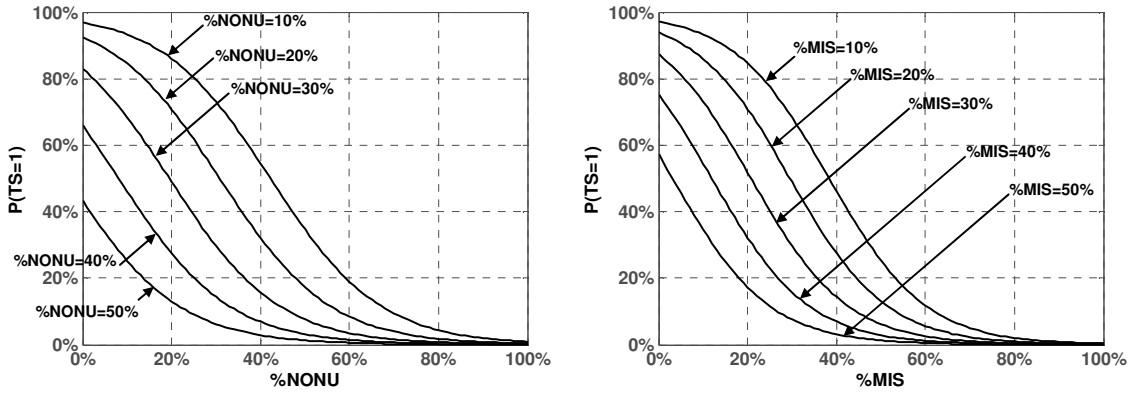


Figure 21. Probability of task success as a function of the misunderstanding (%MIS) and non-understanding (%NONU) error rates in the Let's Go! Public system

To address the first question, we proposed the use of a four-level model for error source analysis, inspired by Clark's model of grounding in conversation [27]. Using the proposed scheme, we conducted empirical investigations of the main sources of understanding-errors in two spoken dialog systems that operate in different domains and have different interaction styles: RoomLine and Let's Go! Public. The results corroborate the intuition that the primary source of errors in these interfaces is the speech recognition process. We believe this result holds across most conversational spoken language interfaces. Our analysis also showed that language-domain errors, such as out-of-domain and out-of-grammar utterances, contribute significantly to the total number of misunderstandings and non-understandings. While the distribution of error sources is generally similar across the two domains, domain-specific differences can also be identified. For instance, a significantly larger number of errors in the Let's Go! Public system is caused by adverse environmental conditions.

Additionally, we proposed a data-driven approach for assessing the impact of various types of understanding-errors on global dialog performance. The proposed approach uses a regression model to relate the frequency of understanding-errors to overall dialog performance. Experiments conducted with the RoomLine and Let's Go! Public systems have confirmed that both misunderstandings and non-understandings exert a significant negative impact on task success. The models we constructed allow us to quantify this impact and provide additional insights into the relationship between understanding-errors and overall performance. We have seen that this relationship is often non-linear and that the costs of errors are different across domains. In RoomLine, misunderstandings are more costly than non-understandings, while in the Let's Go! Public system the two types of errors seem to exert a similar effect on task success. This latter result aligns with the more conservative confirmation policy used by the Let's Go! Public system.

The data-driven error-cost assessment methodology we have introduced in this chapter is not limited to inferring the costs for understanding-errors. We believe the approach can be extended to capture costs for other types of errors or actions that the system engages in. In Chapter 7, we ex-

tend this methodology by introducing state distinctions: error-costs are different not only across dialog systems, but also across different dialog states within the same system; we then use the derived costs to adjusted rejection thresholds in a principled manner, on a state-by-state basis.

In conclusion, the error-source and error-impact analysis methodologies we have described in this chapter have confirmed a number of previous intuitions: speech recognition errors are the major source of understanding-errors; both misunderstandings and non-understandings have a significant effect on overall dialog performance; the costs for different types of errors are different across domains. In addition, they have helped us to more precisely quantify and characterize these intuitions and have provided additional insights: for instance, the impact of understanding-errors on overall dialog performance is often non-linear. We believe these analysis techniques can be very useful in guiding system development and optimization efforts.

PART II.

INFRASTRUCTURE

Chapter 3

The RavenClaw dialog management framework

In this chapter, we describe RavenClaw, a plan-based, task-independent dialog management framework. RavenClaw isolates the domain-specific aspects of the dialog control logic from domain-independent conversational skills, and in the process facilitates rapid development of systems operating in complex, task-oriented domains. System developers can focus exclusively on describing the dialog task control logic, while a large number of domain-independent dialog mechanisms (e.g. error handling, timing and turn-taking) are transparently supported and enforced by the RavenClaw dialog engine. To date, RavenClaw has been used to construct and successfully deploy a number of spoken dialog systems spanning different domains and interaction styles. Together with these systems, RavenClaw provides the infrastructure for the error handling research described in the rest of this dissertation.

3.1 Introduction

The dialog manager component plays a central role in any conversational spoken language interface: given the decoded semantic input corresponding to the current user utterance, it determines the next system action. In essence, the dialog manager is responsible for planning and maintaining the coherence, over time, of the conversation. Several tasks must be performed in order to accomplish this goal successfully.

First, the dialog manager must maintain a history of the discourse and use it to interpret the perceived semantic inputs in the current context. Second, a represention (either explicit or implicit) of the system task is typically required. The current semantic input, together with the current dialog state and information about the task to be performed is then used to determine the next system action. As we shall see later, different theories and formalisms have been proposed for making these decisions. In some dialog managers, a predefined universal plan for the interaction exists, i.e. the system actions are predetermined for any given user input. Other systems make certain assumptions about the structure of the interaction and dynamically plan the next move at run-time, based on generic dialog rules. Often, dialog managers have to also interact with various domain and application-specific agents. For instance, a dialog system that assists users in making flight reservations must communicate with a database to obtain information and to perform the required transactions.

In guiding the conversation, the dialog manager must also be aware of, and must implement a number of conversational norms. A first example is timing and turn-taking. Most systems make the assumption that the two participants in the conversation make successive contributions to the dialog. More complex models need to be developed to support barge-ins, backchannels, or multi-participant conversation. Another example is error handling. In speech-based conversational systems, the dialog management component must be able to take into account the underlying uncertainties in the recognition results and plan the conversation accordingly. Unless robust mechanisms for detecting and recovering from errors are present, speech recognition errors can lead to complete breakdowns in interaction. Other generic conversational skills include the ability to respond appropriately to various requests like "can you repeat that?", "wait a second", etc.

In this chapter, we describe RavenClaw, a plan-based dialog management framework we have developed to provide the infrastructure for the error handling research program outlined in the introduction. We begin by reviewing some of the current dialog management solutions and outlining the main objectives that have guided the development of RavenClaw in this section. Then, in section 3.2, we describe the overall architecture of the RavenClaw dialog management framework, and discuss the various algorithms and data-structures that govern its function and confer the desired properties. In order to build a fully functioning spoken language interface, a number of other components besides a dialog manager are however required: speech recognition, language understanding and generation, speech synthesis, etc. In section 3.3 we give a quick overview of Olympus [12], a collection of freely available dialog system components (and corresponding control logic) that we have used in conjunction with RavenClaw to build and deploy spoken language interfaces. Then, in section 3.4 we describe a number of systems developed using the RavenClaw/Olympus infrastructure. Finally, in section 3.5 we present a number of concludring remarks and discuss directions for further extending the RavenClaw dialog manager and the Olympus framework.

3.1.1 Current dialog management solutions

A number of different solutions for the dialog management problem have been developed to date in the community. Some of the most widely used techniques are: finite-state, form-filling, information-state-update, and plan-based approaches. Each of these approaches makes different assumptions about the nature of the interaction; each has its own advantages and disadvantages.

In a finite-state dialog manager, the flow of the interaction is described via a finite-state automaton. At each point in the dialog, the system is in a certain state (each state typically corresponds to a system prompt). In each state, the system expects a number of possible responses from the user; based on the received response, the system transitions to a new state. To develop a dialog management component for a new application, the system author must construct the corresponding finite state automaton. In theory, the finite-state automaton representation is flexible enough to capture any type of interaction. In practice, this approach is well suited only for implementing relatively simple systems that retain the initiative throughout the conversation. In these cases, the finite-state automaton representation is very easy to develop, interpret, and maintain. However, the finite-state representation does not scale well for more complex applications or interactions. For instance, in a mixed-initiative system (where the user is also allowed to direct and shift the focus of the conversation), the number of transitions in the finite-state automaton grows very large; the representation becomes difficult to handle. Representative examples of this approach are the industry standard VoiceXML [126], the CSLU dialog management toolkit [29, 125], and Nuance's SpeechObjects [78].

Another dialog management technology, especially useful in information access domains is form-filling (also known as slot-filling). In this case the basis for representing the system's interaction task is the form (or frame). A form consists of a collection of slots, or pieces of information to be collected from the user. For instance, in a train schedule information system, the slots might be the departure and arrival city, the travel date and time. Each slot has an associated system prompt that will be used to request the corresponding information from the user. Typically, an action is associated with each form, for instance access to the schedule database in the train system. The system

guides the dialog such as to collect the required slots from the user (some of the slots might be optional); the user can also take the initiative and provide information about slots that the system has not yet asked about. Once all the desired information is provided, the system performs the action associated with the form. The system's interaction task may be represented as a collection of chained forms, with a specied logic for transitioning between these forms. In comparison with the finite-state representation, the form-filling approach makes stronger assumptions about the nature of the interaction task, and in the process allows system authors to more easily specify it. As we have mentioned before, this approach is well-suited in information access domains where the user provides some information to the system, and the system accesses a database or performs an action based on this information. However, the approach cannot be easily used to construct systems in domains with different interaction styles: tutoring, guidance, message delivery, etc. A representative example of the form-filling approach is Phillips' SpeechMania system [3].

A third dialog management approach that has recently received a lot of attention and wide adoption in the research community is information-state-update (ISU). In this approach the interaction flow is modeled again as a sequence of states. However, these states are not explicitly represented like in the finite-state approach. Rather, the system state (also known as information-state) is a data structure that contains information accumulated throughout the discourse. In addition, a set of information-state update rules govern how the system updates its state based on the perceived user inputs. A potential drawback of this approach is that, as the set of update rules increases, interactions between these rules and their overall effects become more difficult to anticipate. At the same time, the ISU approach allows for a high of flexibility in managing the interaction. Different ISU systems can capture different information in the state, and implement different linguistic theories of discourse in the state-update rules. Representative examples of the ISU approach include the TrindiKit dialog move engine [64, 124] and DIPPER [17].

The fourth dialog management technology we have mentioned are plan-based approaches. In this case, the system models the goals of the conversation, and uses a planner to guide the dialog along a path towards these goals. These systems reason about user intentions, and model relationships between goals and subgoals in the conversation, and the conversational means for achieving these goals. As a consequence, they require more expertise from the system developer, but can enable the development of more complex interactive systems. Examples include the TRAINS and TRIPS systems [34], and Collagen [94, 95]. The RavenClaw dialog manager we describe in the rest of this chapter falls in this last category.

3.1.2 RavenClaw

The RavenClaw dialog management framework was developed as a successor of the earlier Agenda dialog management architecture used in the CMU Communicator project [101]. The primary objective was to develop a robust platform for research in dialog management and conversational spoken language interfaces. In support of this goal, we identified and pursued several desirable characteristics:

Task-independence. The framework should provide a clear separation between the domain-specific aspects of the dialog control logic and domain-independent, reusable dialog control mechanisms. This decoupling will significantly lessen the system development effort and promotes reusability of various solutions and components. System authors focus exclusively on describing the domain-specific aspects of the dialog control logic, while a reusable, domain-independent dialog engine transparently supports and enforces a number of generic conversational skills (e.g. error handling, timing and turn-taking, context establishment, help, etc.)

Flexibility. The framework should accommodate a wide range of application domains and interaction styles. Some dialog management formalisms are more suited for certain types of applications. For instance, form-filling approaches are typically well-suited in information access domains; however, the form-filling paradigm does not easily support the development of a tutoring application. The RavenClaw dialog management framework uses a hierarchical plan-based formalism to rep-

resent the dialog task. We have found that this representation provides a high degree of flexibility, while also leading to good scalability properties. To date, RavenClaw has been used to construct over a dozen dialog systems spanning different domains and interaction styles: information-access, guidance through procedural tasks, message-delivery, comman-and-control, web-search, scheduling.

Transparency. The framework should provide access to detailed information about each of its internal subcomponents, states and run-time decisions. RavenClaw supports configurable multiple-stream logging: each of its subcomponents provides generates a rich log stream containing information about internal state, computations, and decisions made. Furthermore, a number of task-independent data analysis and visualization tools have been developed.

Modularity/reusability. Specific functions, for instance dialog planning, input processing, output processing, error- handling, etc. should be encapsulated in subcomponents with well-defined interfaces, that are decoupled from the domain-specific dialog control logic. RavenClaw users are able to inspect and modify each of these components individually, towards their own ends. Modularity promotes the reusability and portability of the developed solutions. For instance, error handling strategies developed in the context of a system that helps users make conference room reservations [14] have been later plugged into a system that provides bus schedule information [89]. The RavenClaw dialog management framework was adapted to work with different types of semantic inputs (Phoenix [131] and Gemini [33]) by simply overwriting the input processing class.

Scalability. The framework should support the development of practical, real-world spoken language interfaces. While simple, well-established approaches such as finite-state call-flows allow the development of practical, large-scale systems in information access domains, this is usually not the case for frameworks that provide the flexibility and transparency needed for research. At the same time, a number of relevant research questions do not become apparent until one moves from toy systems into large-scale applications. In RavenClaw, the hierarchical plan-based representation of the domain-specific dialog control logic confers good scalability properties, while at the same time it does not sacrifice flexibility or transparency.

Open-source. Together with a number of end-to-end spoken dialog systems developed withing this framework, complete source code for RavenClaw has been released under an open-source licence [90]. We hope this will foster further research and contributions from other members of the research community.

To date, RavenClaw has been used to construct a number of applications spanning multiple domains and interaction types. Some of these have been deployed into day-to-day use. For instance, RoomLine is a mixed-initiative telephone-based system that helps users make conference room reservations on the CMU campus [14]. The Let's Go! Public bus information system [87-89] provides bus route and schedule information in the greater Pittsburgh area; this system is connected to the Port Authority of Allegheny County customer service line during non-business hours and receives 50-60 calls per night. ConQuest [11] is a spoken dialog system that provides schedule information during technical conferences; this system has been recently deployed during the Interspeech'06 and IJCAI'07 conferences. Together with these systems, the RavenClaw dialog management framework provides the infrastructure for the error handling work described in the rest of this dissertation. In addition, RavenClaw also provides the basis for a number of additional research projects, such as multi-particiapant conversation [47] and fine-grained timing and turn-taking [86].

3.2 The RavenClaw dialog management architecture

In this section we describe the architecture of the RavenClaw dialog management framework, and the various data-structures and algorithms that govern its operation.

3.2.1 A top-level architectural view

RavenClaw is a two-tier dialog management architecture that enforces a clear separation between the domain-dependent and the domain-independent aspects of dialog control – see Figure 22. The do-

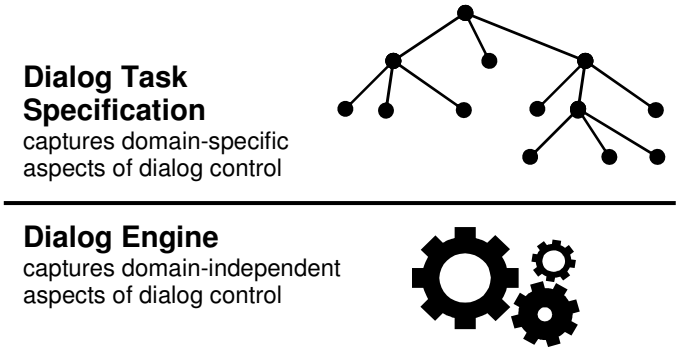


Figure 22. RavenClaw - a two-tier dialog management architecture

main-specific aspects of the dialog control logic are captured by the **dialog task specification**, essentially a hierarchical-plan for the interaction, provided by the system author. A reusable, domain-independent **dialog engine** manages the conversation by executing the given dialog task specification. In the process, the dialog engine also contributes a basic set of **domain-independent conversational strategies** such as error handling, timing and turn-taking behaviors, and a variety of other universal dialog mechanisms, such as help, repeat, cancel, suspend/resume, quit, start-over, etc.

The decoupling between the domain-specific and domain-independent aspects of dialog control significantly lessens the system authoring effort. System developers can focus exclusively on describing the dialog task control logic, while a large number of domain-independent dialog mechanisms are transparently supported and enforced by the dialog engine. Consider for instance error handling. System developers construct a dialog task specification under the assumption that inputs to the system will always be perfect, therefore ignoring the underlying uncertainties from the speech recognition channel. The responsibility for ensuring that the system maintains accurate information and that the dialog advances normally towards its goals is delegated to the dialog engine (how exactly the engine makes this happen makes the subject of the next chapter). This decoupled approach significantly lessens the authoring effort, promotes portability and reusability, and ensures a certain degree of uniformity and consistency in behavior both within and across tasks.

In the rest of this section, we describe in more details the internals of the RavenClaw dialog management architecture. We begin by discussing the dialog task specification language in the next subsection. Then, in subsection 3.2.3, we describe the algorithms and data-structures which govern the RavenClaw dialog engine. Finally, we discuss the set of task-independent conversational strategies automatically supported by the dialog engine.

3.2.2 The dialog task specification

The dialog task specification captures the domain-specific aspects of the dialog control logic. Developing a new dialog manager using the RavenClaw framework therefore amounts to writing a new dialog task specification. We begin with a high-level overview.

The dialog task specification describes a hierarchical plan for the interaction. More specifically, the dialog task specification consists of a tree of **dialog agents**, where each agent is responsible for handling a subpart of the interaction. For instance, Figure 23 depicts the top portion of the dialog task specification for RoomLine, a spoken dialog system which can assist users in making conference room reservations (more details about this system are presented in subsection 3.4.1). The root node subsumes several children: `Login`, which identifies the user to the system, `GetQuery`, which obtains the time and room constraints from the user, `GetResults`, which executes the query against the backend, and `DiscussResults` which presents the obtained results and handles the forthcoming negotiation for selecting the conference room that best matches the user's needs. Moving one level

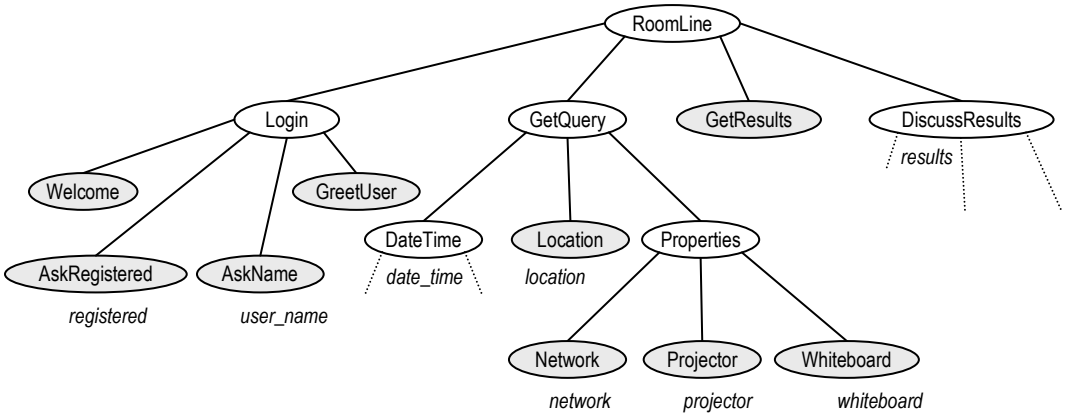


Figure 23. A portion of the dialog task tree for the RoomLine system

deeper in the tree, the `Login` agent decomposes into `Welcome`, which provides a short welcome prompt, `AskRegistered` and `AskName`, which identify the user to the system, and finally `GreetUser`, which sends a greeting to the user.

The dialog agents in a dialog task specification fall into two categories: **fundamental dialog agents**, shown grayed in Figure 23, and **dialog-agencies,** shown in clear in Figure 23. The **fundamental dialog agents** are located at the terminal positions in the tree (e.g. `Welcome`, `AskRegistered`) and implement atomic dialog actions, or dialog moves. There are four types of fundamental dialog agents: **Inform** – produces an output (e.g. `Welcome`), **Request** - requests information from the user (e.g. `AskRegistered`), **Expect** - expects information from the user, but without explicitly requesting it (e.g. `Projector`) and **Execute** - performs a domain-specific operation, such as database access (e.g. `GetResults`). The **dialog-agencies** occupy non-terminal positions in the tree (e.g. `Login`, `GetQuery`); their purpose is to control the execution of their subsumed agents, and encapsulate the higher level temporal and logical structure of the dialog task.

Each dialog agent implements an `Execute` routine, which is invoked at runtime by the dialog engine. The `Execute` routine is specific to the agent type. For example, inform-agents generate an output when executed, while request-agents generate a request but also collect the user's response. For dialog-agencies, the `Execute` routine is in charge of planning the execution of their subagents. Besides the `Execute` routine, each dialog agent can define preconditions, triggers, as well as success and failure criteria. These are taken into account by the dialog engine and parent dialog-agencies while planning the execution of the various agents in the tree.

If the dialog agents are the fundamental execution units in the RavenClaw dialog management framework, the data that the system manipulates throughout the conversation is encapsulated in **concepts.** Concepts can be associated with various agents in the dialog task tree (e.g. `registered` and `user_name` in Figure 23), and can be accessed and manipulated by any agent in the tree. Several basic concept types are predefined in the RavenClaw dialog management framework: Boolean, string, integer and float. Additionally, the framework provides support for more complex, developer-defined concept types such as (nested) structures and arrays. Internally, the "value" for each concept is represented by a set of value/confidence pairs (e.g. `city_name = {Boston/0.35; Austin/0.27}`). The dialog engine can therefore track multiple alternate hypotheses for each concept, and can quantitatively capture the level of uncertainty in each hypothesis. Additionally, each concept also maintains the history of previous values, as well as information about the grounding state, when the concept was last updated, etc.

To summarize, the dialog task tree describes an overall hierarchical plan for the interaction. However, this developer-specified plan does not prescribe a fixed order for the execution of the various dialog agents (as might be found in a directed dialog system). When the dialog engine executes a given dialog task specification, a particular trace through this hierarchical plan is followed,

based on the user inputs, the encoded domain constraints and logic, as well as the various execution policies in the dialog engine. This type of hierarchical task representation has been used for tasks execution in the robotics community. More recently, this formalism has gained popularity in the dialog management community. Other examples besides RavenClaw include the use of a tree-of-handlers in Agenda Communicator [101], activity trees in WITAS [66] and recipes in Collagen [94, 95]. In the context of spoken dialog systems, hierarchical plan-based representations present several advantages. Most goal-oriented dialog tasks have an identifiable structure which naturally lends itself to a hierarchical description. The subcomponents are typically independent, leading to ease in design and maintenance, as well as good scalability properties. The tree representation captures the nested structure of dialog and thus implicitly represents context (via the parent relationship), as well as a default chronological ordering of the actions (i.e. left-to-right traversal). Finally, the tree structure can be extended at run-time, and allows for the dynamic construction of dialog structure, a very useful feature in certain types of tasks.

In the next three subsections, we describe in more detail the various components of the dialog task specification. We begin by introducing the dialog task specification language in the next subsection. Then, in subsection 3.2.2.2, we describe in more detail the four types of fundamental dialog agents, and the dialog-agencies. Finally, in subsection 3.2.2.3 we describe in more detail the concept representation and the associated concept operators.

3.2.2.1 The RavenClaw dialog task specification language

The dialog task specification is described by the system author using an extension of the C++ language constructed around a set of predefined macros. Figure 24 illustrates a portion of the dialog task specification for the RoomLine system, corresponding to the `Login` sub-tree.

Each agent in the dialog task tree is specified using a define agent directive (e.g. `DEFINE_AGENCY`, `DEFINE_REQUEST_AGENT`, etc.) For instance, the lines from 1 to 10 define the `Login` dialog-agency. This agency stores a Boolean concept which indicates whether the user is registered with the system or not (`registered`) and a string concept that will contain the user's name (`user_name`). The agency has 4 subagents (`Welcome`, `AskRegistered`, `AskName` and `GreetUser`), and succeeds when the `GreetUser` agent has completed. Lines 13-15 define the `Welcome` inform-agent. This agent sends a non-interruptible output prompt that will welcome the user to the system. Next, lines 17-20 define the `AskRegistered` request-agent. When executed, this agent asks whether the user is registered with the system or not. It expects a `[Yes]` or a `[No]` semantic answer, and, upon receiving such an answer, it will fill in the registered concept with the appropriate value (e.g. true or false). Similarly, the `AskName` agent defined in lines 22-26 asks for the `user_name` concept. Note that this agent also has a precondition, i.e. that the `registered` concept is true. If the user answered that she is not registered with the system this agent will be skipped during execution. Finally, lines 28-31 define the `GreetUser` inform-agent. This agent sends out a greeting prompt; the values of the `registered` and `user_name` concepts are also sent as parameters to the language generation module.

The various macros shown in this example (e.g. `DEFINE_AGENCY`, `DEFINE_REQUEST_AGENT`, `REQUEST_CONCEPT`, `GRAMMAR_MAPPING`, etc) are expanded at compile-time by the C++ preprocessor and a class is generated for each defined dialog agent. The directives and parameters specified for each agent (e.g. `PROMPT`, `REQUEST_CONCEPT`, `GRAMMAR_MAPPING`, etc.) are expanded into methods that overwrite virtual methods from the base class, in effect customizing the agent to implement the desired behavior.

Next, we describe in more detail the dialog task specification language, and the various types of agents and concepts that can populate the dialog task tree. This description is not exhaustive, but rather it is meant to illustrate some of the main aspects of the dialog task specification language. The reader who is not interested in the dialog task specification language can safely skip to section 3.2.3, where we discuss the algorithms that govern the RavenClaw dialog engine.

```
DEFINE_AGENCY( CLogin,
  DEFINE_CONCEPTS(
    BOOL_USER_CONCEPT( registered, “default” )
    STRING_USER_CONCEPT( user_name, “default” )
  DEFINE_SUBAGENTS(
    SUBAGENT( Welcome, CWelcome, “” )
    SUBAGENT( AskRegistered, CAskRegistered, “default” )
    SUBAGENT( AskName, CAskName, “default” )
    SUBAGENT( GreetUser, CGreetUser, “” ))
  SUCCEEDS_WHEN(COMPLETED(GreetUser))
)

DEFINE_INFORM_AGENT( CWelcome,
  PROMPT(“:non-interruptible inform welcome”)
)

DEFINE_REQUEST_AGENT( CAskRegistered,
  REQUEST_CONCEPT(registered)
  GRAMMAR_MAPPING(“[Yes]>true, [No]>false”)
)

DEFINE_REQUEST_AGENT( CAskName,
  PRECONDITION(IS_TRUE(registered))
  REQUEST_CONCEPT(user_name)
  GRAMMAR_MAPPING(“[Identification.user_name]”)
)

DEFINE_INFORM_AGENT( CGreetUser,
  PROMPT(“inform greet_user <registered <user_name”)
)
```

Figure 24. A portion of the dialog task specification for the RoomLine system

3.2.2.2 Dialog agents

The dialog agents in the dialog task specification fall into two main categories: fundamental dialog agents (Inform, Request, Expect, Execute), located at the terminal positions in the tree, and dialog-agencies located at non-terminal positions.

A number of methods and parameters are common to all agent types.

Each agent can define a **precondition** (via the `PRECONDITION` macro). Under the default execution policy (see the Dialog-agencies below), agents are planned for execution only when their preconditions hold. By default, unless otherwise specified by the system developer through the `PRECONDITION` macro, the precondition value is true. For instance, in the example from Figure 24, the `AskName` agent has a precondition that the `registered` concept is `true`.

Each agent can also define a **success** (`SUCCEEDS_WHEN`) and a **failure criterion** (`FAILS_WHEN`). An agent is considered completed by the dialog engine when either one of these criteria are met. The default success criterion is different for each type of agent. By default, Inform and request-agents succeed as soon as they have executed once. Request-agents succeed when there is a new, grounded value for the concept they request. Dialog-agencies succeed when all their subagents have completed. Finally, expect-agents do not have a success criterion since they never get executed (see more in the expect-agents subsection below). In the example from Figure 24, the `Login` agency overwrites the default success criterion: this agency succeeds as soon as the `GreetUser` agent has completed its execution. The default failure criterion on all agents is that the number of execution attempts exceeds a maximum execution attempt counter (which can be controlled via the `MAX_ATTEMPTS` directive.)

Each agent can also define a **trigger condition** (`TRIGGERED_BY`) and/or a **trigger com-**

mand (TRIGGERED_BY_COMMANDS). The trigger conditions and trigger commands are checked by the dialog engine at each turn in the dialog. When the trigger condition becomes true, or when the current semantic input matches the trigger command, the dialog engine will shift the focus to this agent, by placing it on top of the dialog stack (more details about the focus shift mechanism are discussed later, in section 3.2.3.1).

Additionally, a number of operations can be performed on dialog agents, via a set of helper methods:

- The **finish** operation (via the FINISH(agent) directive) forces a dialog agent to complete and eliminates it from the dialog stack.
- The **reset** operation (via the RESET(agent) directive) performs a re-initialization of the agent. All the subagents are also reset, and all concepts held by the agent are cleared. The agent is marked as not-completed.
- The **reopen** operation (via the REOPEN(agent) directive) is similar to the reset operation. However, the concepts are reopened rather than cleared (i.e. the current value of the concept is pushed into history, and the concept is again empty – see more details in subsection 3.2.2.3).
- The **reopen topic** operation (via the REOPEN_TOPIC(agent) directive) is similar to reopening an agent, but it leaves the concepts untouched.

Finally, each agent can also define a number of call-back methods:

- ON_CREATION is executed immediately after the agent is created.
- ON_INITIALIZATION is executed when the agent is initialized. Agents are initialized immediately after they are created, but also when they are reset or reopened (see the RESET and REOPEN operations below).
- ON_DESTRUCTION is executed right before the agent is destroyed.
- ON_REOPEN is executed when an agent is reopened for conversation (see REOPEN_AGENT)
- ON_COMPLETION is executed when the agent completes and is eliminated from the dialog stack by the dialog engine; system authors can use this macro to write side-effect code that will be invoked upon completion of the agent.

Each agent in the dialog task tree can be referenced by the other agents using the A() function call with a relative or absolute tree path to the agent. For instance A(/RoomLine/Login) refers to the Login dialog-agency in the example from Figure 23, by using an absolute path. Similarly, in the context of the AskName agent, A(../AskRegistered) refers to the sibling agent AskRegistered.

In the next subsections, we describe some of the main methods and parameters that are specific to the four types of fundamental dialog agents: Inform, Request, Execute, and Expect.

§ Inform-agents

Inform-agents are defined via the DEFINE_INFORM_AGENT macro:

```
DEFINE_INFORM_AGENT(CAgentTypeName,
       {directive}
)
```

The role of inform-agents is to issue output prompts. The prompt for each inform-agent is specified through the PROMPT directive (see lines 14 and 29 in Figure 24.) The prompt specification obeys the following syntax:

```
prompt    ::= <act> <object> {parameter}
parameter ::= [attr] '<' <concept> |
              attr '=' value
```

The act and the object uniquely identify the prompt for the language generation module. A number of acts are predefined and commonly used in the RavenClaw framework: inform, request, explicit_confirm, implicit_confirm, establish_context. The system developer can extend this list, while ensuring that the language generation module handles the new prompts accord-

ingly.

The prompt parameters provide the language generation module with an attribute-values list for the prompt. This list is used to populate the language generation templates with values. An attribute for a prompt can be declared by simply specifying a concept name or referent (the attribute name will be set to the concept name, and the attribute value to the concept value), or by specifying an attribute name and a constant value. For instance, the prompt on line 29 in Figure 24 is:

```
inform welcome <registered <user_name
```

The language generation module receives as parameters the `registered` and `user_name` concepts and can use them to construct an appropriate prompt. For instance, if `registered=true` and `user_name="John Doe"`, the natural language generation might generate "Hi John Doe!". Alternatively if `registered=false` and `user_name=""`, the natural language generation module might generate "Hello, guest user! If you would like to register with the system, please send an email to roomline@cs.cmu.edu."

A number of additional flags control how the prompts are rendered:

- `:non-listening` – the decoder will be set in non-listening mode for the duration of the prompt.
- `:non-interruptible` – the barge-in mechanism will be disabled for the duration of the prompt.
- `:non-repeatable` – this prompt will not be repeated by the output manager, if the user asks the system to repeat.
- `:<device>` – specifies the output device to which the prompt will be sent. By default, prompts are sent to the natural language generation device. However system developers can specify additional output devices (e.g. a GUI in a multimodal system.)

§ Request-agents

Request-agents are defined via the `DEFINE_REQUEST_AGENT` macro:

```
DEFINE_REQUEST_AGENT(CAgentTypeName,
       {directive}
)
```

The role of request-agents is to request and acquire concept values from the user.

To do so, a request-agent declares the concept it requests via the `REQUEST_CONCEPT` macro, and the semantic grammar slots it expects to hear from the user via the `GRAMMAR_MAPPING` macro. For instance, here is again the `AskRegistered` request-agent from the previous example:

```
DEFINE_REQUEST_AGENT( CAskRegistered,
  REQUEST_CONCEPT(registered)
  GRAMMAR_MAPPING("[Yes]>true, [No]>false")
)
```

The agent requests the `registered` Boolean concept (which is defined and stored in the parent agent, `Login`). The grammar mapping describes what the agent expects to hear in the user response, and how the values from the semantic parse will be mapped into a corresponding value for the concept. In this case, the agent expects to hear a `[Yes]` or a `[No]` answer. If it hears a `[Yes]` answer, it will update the registered concept with the value `true`; if it hears a `[No]` answer, it will update the registered concept with the value `false`. More generally, the syntax for the grammar mapping definition is:

```
grammar_mapping ::= {gm_element,}
gm_element      ::= [gm_scope] '['<grammar_slot_name>']'
[ '>' (value | ':'binding_filter_name) ]
gm_scope        ::= '!' | '@' | '*' | '@('{<agent_referent>;}')'
```

Here are four other example grammar mappings:

1. `[Identification.user_name]`
2. `![Yes]>projector, [Projector]>projector`
3. `[DateTime.date_relative]>:datetime`
4. `@(/RoomLine/AnythingElse)[NeedRoom.date_time]>:datetime`

A grammar mapping is generally defined as a comma separated list of grammar mapping elements. Each element specifies one semantic grammar slot (e.g. `[Identification.user_name]`, `[Yes]`, etc.) During the input processing phase, the dialog engine will look for that slot in the current input. If the slot is found, the value for the slot (the corresponding string is the input) is used to update the requested concept. For instance, if the recognition result is *"My name is John Doe"* we obtain the parse:

```
[Identification] (my name is [user_name] (John Doe))
```

In this case, according to grammar mapping (1), the string *"John Doe"* will be used to update the requested concept.

The `>value` postfix construct allows system developers to perform a very simple normalization of the input string. For instance, for the user response *"That's right"*, we obtain the parse:

```
[Yes] (that's right)
```

In this case, according to grammar mapping (2), the value `projector` will be used to update the requested concept.

More sophisticated processing of the input can be accomplished by using **binding filters**. A binding filter is a developer-defined function that is applied to the parse string to construct a concept value. For instance, for grammar mapping 4, if the user responds *"I need a room tomorrow"*, we obtain the parse:

```
[NeedRoom](i need a room) [DateTime]([date_relative](tomorrow))
```

In this case, according to grammar mapping (3), the string *"tomorrow"* is passed to the `datetime` binding filter. This filter will process the string and return an actual date-time value, such as 2006-03-22, that will be used to update the requested concept.

During the input processing phase, each defined expectation can be either **active** or **closed**. A request-agent can collect information from the input only through active expectations. The system developer can control at what times expectations are active by using a scope operator in front of the grammar slot (e.g. `!`, `*`, or `@`). For clarity purposes, we will discuss these scope operators later, in section 3.2.3.2, when we describe the input processing algorithms in the dialog engine. Additionally, the system author can control when expectations are active by specifying a Boolean condition with an `EXPECT_WHEN` directive.

Unless otherwise specified by the system developer, the precondition for a request-agent is that no value is available for the requested concept. The success criterion is that the requested concept has been updated.

Apart from the requested concept and the grammar mapping, system developers can also specify the **prompt** for the request-agent via the `PROMPT` directive. The prompt syntax is the same as the one presented above for the inform-agents. If a request-agent does not specify a prompt, then a `{request <concept_name>}` prompt will be generated by default, where `concept_name` is the name of the requested concept (declared via `REQUEST_CONCEPT`). In the language generation module, system authors may specify four different versions of the request prompt: (1) `default`, (2) `explain_more`, (3) `what_can_i_say`, and (4) `timeout`. Normally, the `default` version is invoked by the dialog manager. The `explain_more` version contains a more comprehensive version of the prompt, which is used by the dialog engine on certain types of help requests (see subsection 3.2.4.1). Similarly, the `what_can_i_say` version contains information about how the user could respond to the system request; this prompt is also invoked to provide help to the users. Finally, the `timeout` version is used when the system re-prompts the user if a timeout has elapsed (see subsection 3.2.4.1)

System developers may specify for each request-agent a **rejection threshold,** via the `REJEC-`

TION_THRESHOLD directive, and a **timeout period**, via the TIMEOUT_PERIOD directive. If the confidence score of the input is below the rejection threshold when the agent is in focus, the input is rejected and no concept updates happen. Finally, if the timeout period elapses and no user input is received, the system internally generates a timeout event, which is captured and handled accordingly by the timeout handling agency (see section 3.2.4.1).

§ Expect-agents

Expect-agents are defined via the DEFINE_EXPECT_AGENT directive:

```
DEFINE_EXPECT_AGENT( CAgentTypeName,
       {directive}
)
```

Expect-agents are used to acquire information from the user input, without explicitly requesting for this information (as the request-agents do). The expect-agents are never directly executed by the dialog engine, but they declare their expectations for the user input and participate in the input processing phase (described in detail in subsection 3.2.3.2). Just like request-agents, the expect-agents declare the concepts they expect through the EXPECT_CONCEPT directive, and the semantic grammar slots they expect to hear through the GRAMMAR_MAPPING directive.

§ Execute-agents

Execute-agents are defined via the DEFINE_EXECUTE_AGENT directive:

```
DEFINE_EXCUTE_AGENT( CAgentTypeName,
       {directive}
)
```

The role of execute-agents is to implement various domain-specific operations, such as calls to various back-end modules (e.g. database, back-end reasoning component, application interface, etc). By default, the success condition for an execute-agent is that it has executed once.

Calls to back-end modules can be declared via the CALL directive. The syntax is as follows:

```
call      ::= <module.function> query=<query_name>
              {parameter} [:non-blocking]
parameter ::= [attr] '<' <concept> | [attr] '>' <concept> |
              attr '=' value |
```

module.function identifies the external component and service that is invoked. query_name defines the type of query. The parameter list is very similar to the one used in the prompt definitions. The only exception is that now return values can be collected and used to update concepts using the attr > concept construct. Finally, the :non-blocking flag is used to specify calls for which the dialog manager should not expect an immediate answer.

More generally, system developers can overwrite the entire Execute routine for execute-agents via the EXECUTE directive. By default, the Execute routine invokes the back-end calls specified through the CALL directive.

§ Dialog-agencies

Dialog-agencies are defined via the DEFINE_AGECY directive:

```
DEFINE_AGECY( CAgentTypeName,
       {directive}
)
```

For example, here is again the Login dialog-agency from Figure 24:

```
DEFINE_AGENCY( CLogin,
  DEFINE_CONCEPTS(
    BOOL_USER_CONCEPT( registered, "default" )
    STRING_USER_CONCEPT( user_name, "default" )
  DEFINE_SUBAGENTS(
    SUBAGENT( Welcome, CWelcome, "" )
```

```
        SUBAGENT( AskRegistered, CAskRegistered, "default" )
        SUBAGENT( AskName, CAskName, "default" )
        SUBAGENT( GreetUser, CGreetUser, "" ))
      SUCCEEDS_WHEN(COMPLETED(GreetUser))
)
```

Dialog-agencies are located at non-terminal position in the dialog task tree. Each dialog-agency handles a subpart of the interaction; their role is to plan and coordinate the execution of their subagents. This sub-task planning problem is currently handled by combining a set of simple **execution policies** (i.e. left-to-right traversal of subagents), while checking the preconditions that each agent holds. More sophisticated execution policies can be manually specified by the system developers by overwriting the `EXECUTE` routine for the corresponding dialog agency. In all the systems built to date using the RavenClaw dialog management framework, the simple planning method described above has sufficed. At the same time, the framework leaves open the possibility of developing more advanced execution/planning algorithms.

The subagents for a dialog-agency are defined via the `DEFINE_SUBAGENTS`, and `SUBAGENT` directives, as illustrated above. The `SUBGENT` directive takes three parameters: the name of the subagent, the type of the subagent and the type of error handling model to be used in conjunction with this subagent. The error handling models implement the error handling behaviors associated with a particular dialog agent, and are discussed in more detail in Chapter 4.

Unless otherwise specified by the system developer, the success criterion for a dialog-agency is that all its subagents have completed successfully. However, this condition is often overwritten, as in the example shown above. In this example, the `Login` agency succeeds as soon as the `GreetUser` agent has completed, regardless of the state of its other subagents.

3.2.2.3 Concepts

Having introduced the types of dialog agents that populate the dialog task tree, we now turn our attention to **concepts**. Concepts encapsulate the information (i.e. the data) that the system manipulates throughout the conversation. They are defined and stored in various dialog-agents (usually in dialog-agencies), via the `DEFINE_CONCEPTS` macro. For instance, in the example discussed above, the `Login` agency defines two concepts: `registered`, which is a Boolean concept and `user_name`, which is a string concept. Like agents, concepts can be referenced from any dialog agent in the task tree by using the `C()` function, with a relative or with an absolute path (e.g. `../Login/registered` or `/RoomLine/Login/registered`).

Each concept has a predefined type. A number of basic concept types, such as Boolean, integer, string and float are predefined in the RavenClaw dialog management framework. Additionally, system developers can define custom extended types such as frames, structures and arrays.

Each concept maintains:

- a set of value/confidence pairs, as illustrated in Figure 25; this representation allows the dialog manager to monitor the uncertainty in the information it manipulates, and to engage in concept-level error handling strategies such as explicit and implicit confirmation, disambiguation, etc.
- a concept error handling model that implements the error handling behaviors associated with the concept (in the next chapter, we will describe extensively the error handling architecture in the RavenClaw dialog management framework, including the concept error handling models.)
- a history of previous concept values for the concept.
- information about when the concept was last updated.
- information about whether or not the concept was conveyed to the user.
- flags indicating whether the concept was grounded, and which hypotheses were explicitly or implicitly confirmed.
- information about the prior likelihood of each potential concept value.

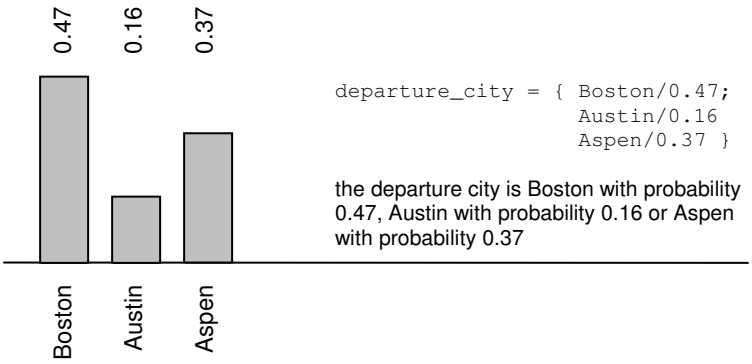


Figure 25. Value/confidence concept representation

- information about the confusability of each potential concept value.

Additionally, a number of concept operators are available:

- assignment operators are used to assign concept values from other concepts or variables.
- comparison operators are use for comparing the values of different concepts of the same type.
- history operators are used to access and manipulate the concept history values.
- belief updating operators. Throughout the interaction with the users, concepts are updated with information from the subsequent user inputs. The belief updating operation takes into account (1) the current set of hypotheses for a concept, with their respective confidence scores, (2) the set of current hypotheses heard from the recognizer, and (3) other contextual information, and generates as a result a new set of value/confidence pairs reflecting the new system belief. Performing accurate belief updating constitutes one of the main foci of the research described in this dissertation, and is discussed extensively in Chapter 6. For now, it suffices to say that RavenClaw supports both heuristic belief updating rules (typically used in most spoken language interfaces), as well as data-driven, model-based belief updating approach.
- other operators, allowing access to various concept attributes and flags (e.g. concept name, whether or not the concept was conveyed to the user, whether or not the concept is grounded, etc.)

Together with the belief updating operators, this rich concept representation provides the necessary infrastructure for performing concept-level error handling in the RavenClaw dialog management. The concept error handling models and the error handling architecture are described in more detail later, in the next chapter. The belief updating mechanisms are discussed in Chapter 6.

3.2.3 The RavenClaw dialog engine

In this section we discuss the algorithms used by the RavenClaw dialog engine to execute a given dialog task specification. The dialog engine algorithms are centered on two data-structures: a **dialog stack**, which captures the discourse structure at runtime, and an **expectation agenda**, which captures what the system expects to hear from the user in any given turn. The dialog is controlled by interleaving **Execution Phases** with **Input Phases** (see Figure 26). During the Execution Phase, dialog agents from the task tree are placed on and executed from the dialog stack, creating in the process the system behavior. During the Input Phase the system uses the expectation agenda to transfer information from the current user input into the concepts defined in the dialog task tree. Below, we describe in more detail each of these two phases.

3.2.3.1 The Execution Phase

During the Execution Phase the RavenClaw dialog engine performs a number of operations in succession (see Figure 26).

First, the dialog engine invokes the `Execute` routine of the agent on top of the dialog stack. The effects of the `Execute` routine are different from one agent-type to another, and have been described previously in Section 3.2.2.2. For instance, inform-agents output a system prompt; request-agents output a system request and then request an Input Phase; dialog-agencies push one of their subagents on the dialog stack. Once the `Execute` routine completes, the control is returned to the dialog engine. If no Input Phase was requested (some agents can make this request upon completing the `Execute` routine), the dialog engine tests the completion conditions for all the agents on the dialog stack. Any completed agents are eliminated from the dialog stack. Next, the dialog engine invokes the Error Handling Decision Process. In this step, the Error Handling Decision Process (described in detail in the next chapter) collects evidence about how well the dialog is proceeding, and decides whether or not to engage in an error handling action. If an error recovery action is necessary, the Error Handling Decision Process dynamically creates and pushes an error handling agency (e.g. explicit confirmation, etc.) on the dialog stack. Finally, in the last stage of the Execution Phase, the dialog engine inspects the focus claims (or trigger) conditions for all the agents in the dialog task tree. If any agents in the task tree request focus, they will be pushed on top of the dialog stack.

To better illustrate the Execution Phase, we will present a step-by-step trace through the execution of the RoomLine dialog task – see Figure 27. The corresponding dialog task tree is also shown in the same figure. At start-up, the dialog engine places the root agent (`RoomLine` in this case) on the dialog stack. Next, the dialog engine goes into an Execution Phase. First, the engine invokes the `Execute` routine for the agent on top of the stack – `RoomLine`. `RoomLine` is a dialog-agency, which, based on its execution policy and on the preconditions of its subagents, decides that it needs to first engage the `Login` agent. It therefore pushes `Login` on the dialog stack (see Figure 27, step 2),

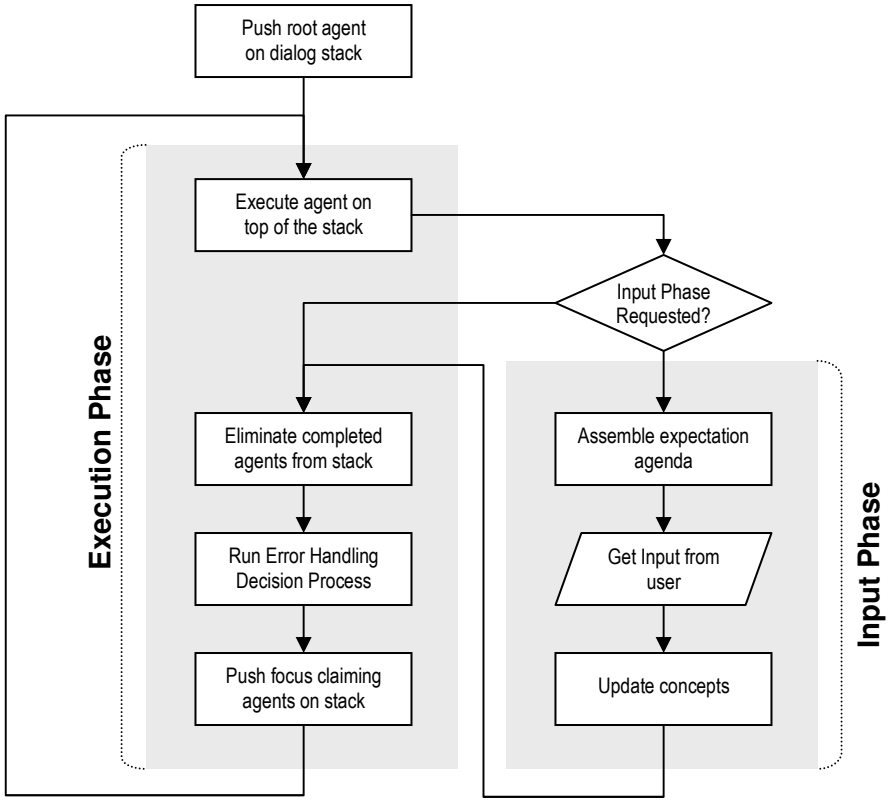


Figure 26. Block diagram for core dialog engine routine

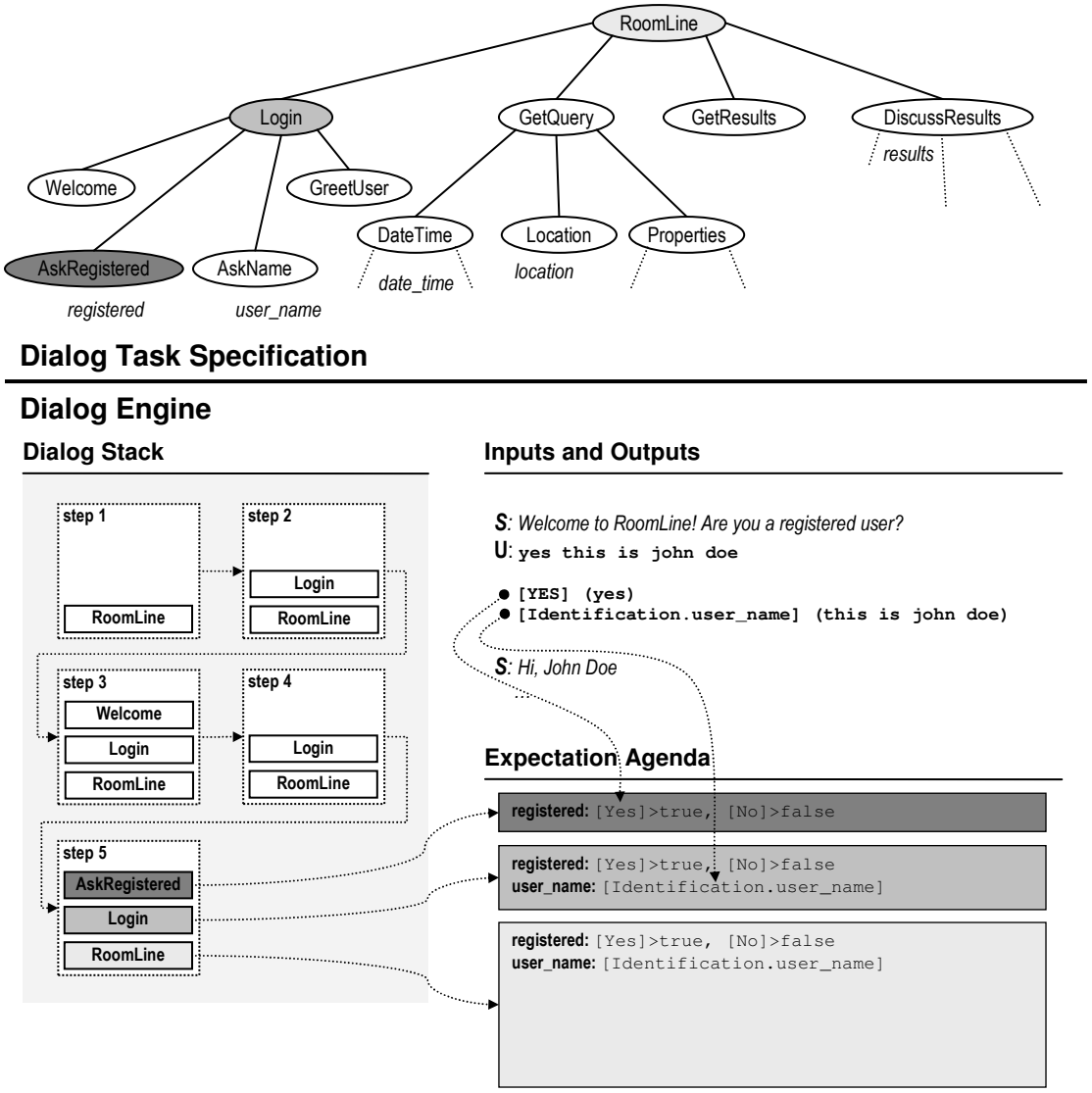


Figure 27. Execution trace through the RoomLine task

and returns the control to the dialog engine. Next the dialog engine pops all completed agents from the dialog stack. Since neither `RoomLine` nor `Login` is yet completed, the dialog engine continues by invoking the error handling decision process. No actions error handling actions are taken in this case[8]. Next the dialog engine inspects the focus claims, but no focus claims are present at this point. The dialog engine therefore engages in a new Execution Phase. This time, `Login` is on top of the stack, so the dialog engine invokes `Login.Execute`. `Login` pushes the `Welcome` agent on the dialog stack and returns the control to the dialog engine (see Figure 27, step 3). Again no agents are completed, no grounding actions are taken and no focus claims are present. Next, the dialog engine executes `Welcome`. This is an inform-agent, which will send out a welcome message to the user. The system says: "Welcome to RoomLine, the conference room reservation assistant." Next, when the dia-

[8] For clarity purposes, we have kept this example simple. For instance, no focus shifts or error handling strategies have been illustrated. In the next section, we will present another example that includes a focus shift. Another more complex execution trace which involves the invocation of various error handling strategies is presented in the next chapter, during the discussion of the error handling architecture.

log engine inspects the completion conditions, it will find that `Welcome` has completed (inform-agents complete as soon as they output the prompt), and it will therefore pop `Welcome` from the execution stack (see Figure 27, step 4). In the next execution phase, `Login.Execute` is invoked again. This time, since the `Welcome` agent is already completed, the `Login` agency will plan the `AskRegistered` agent for execution by pushing it on the dialog stack (see Figure 27, step 5). Again, none of the agents on the stack are completed, no grounding actions are taken and no focus claims are made. When the dialog engine next executes `AskRegistered`, this agent will output a request ("Are you a registered user?"), and then invoke an Input Phase by passing a certain return-code to the dialog engine. We will discuss the Input Phase in the next section.

The dialog stack therefore captures the nested structure of the discourse. Note that the isomorphism between the dialog stack and the task tree is only apparent. There is an essential functional difference between the two structures: the dialog stack captures the temporal and hierarchical structure of the discourse, while the tree describes the hierarchical goal structure of the dialog task. Although in the example discussed above the two structures match, this is not always the case. For instance, if the trigger condition for an agent foo becomes true, the dialog engine will push that agent on top of the dialog stack. The dialog focus will be shifted to that agent. The execution will therefore continue with that agent, and the isomorphism between the stack and the tree will be broken. Once the agent completes and is popped from the stack, we're back to where we were before the focus shift (a concrete example is shown in the next section).

In general, the agent on top of the stack represents the current focus of the conversation, and, the agents below it (which typically sit above it in the tree) capture successively larger discourse contexts. As we have already seen, the dialog stack provides support for maintaining the context during focus shifts and correctly handling sub-dialogs. Additionally the dialog stack is used in to construct of the system's agenda of expectations at every turn, described in the next section.

3.2.3.2 The Input Phase

§ Overview

An Input Phase is invoked each time a request-agent is executed (in the example discussed above, the Input Phase was triggered by the execution of the `AskRegistered` agent). Each Input Phase consists of three stages: (1) assembling the expectation agenda, (2) obtaining the input from the user, and (3) updating the system's concepts based on this input.

First, the system assembles the **expectation agenda**, i.e. a data-structure that describes what the system expects to hear from the user in the current turn. The agenda is organized into multiple levels. Each level corresponds to one of the agents on the dialog stack, and therefore to a certain discourse segment. In the example described above, immediately after the `AskRegistered` agent triggered an Input Phase, the stack contains the `AskRegistered`, `Login` and `RoomLine` agents – see Figure 27, step 5. The dialog engine therefore constructs the first level of the agenda by collecting the expectations from the `AskRegistered` agent. The `AskRegistered` agent expects to hear a value for `registered` concept, in the form of either a `[Yes]` or a `[No]` grammar slot in the input. The second level in the agenda is constructed by collecting the expectations from the next agent on the stack, i.e. `Login`. When an agency declares its expectations, by default it collects all the expectations of its subagents. In this case, the `Login` agency expects to hear both the `registered` concept (from the `AskRegistered` agent), and the `user_name` concept (from the `AskUserName` agent). Finally, the third (and in this case last) level of the expectation agenda is constructed by collecting all the expectations from the `RoomLine` agent. Apart from the `registered` and `user_name` concepts, this last level contains the expectations from all other agents in the dialog task tree. In effect, the levels in the expectation agenda encapsulate what the system expects to hear starting from the currently focused question and moving in larger and larger discourse segments (contexts).

After the expectation agenda has been assembled, the dialog engine waits for an input from the user.

Finally, once the input arrives, the dialog engine engages in a **concept binding** stage. In this

step, the information available in the input is used to update system concepts. The updates are governed by the expectation agenda. The dialog engine performs a top-down traversal of the agenda, looking for matching grammar slots in the user input. Wherever a match is found, the corresponding concept is updated accordingly. For instance, in the example from Figure 27, the recognized user response is *"Yes this is John Doe"*, which parses as:

```
[YES] (yes) [Identification.user_name] (john doe)
```

In this case the `[YES]` slot matches the expectation for the `registered` concept on the first level in the agenda, and the `[Identification.user_name]` slot matches the expectation for the `user_name` concept on the second level in the agenda. These two concepts will be updated accordingly: the `registered` concept will contain `true`, and the `user_name` concept will contain "`john doe`". The concept updating process relies on the belief updating operators, which take into account information about the initial belief about the concept (i.e. the set of alternate hypotheses and their corresponding confidence scores), the user response, as well as the current context (e.g. the last system action, etc).

Once the Input Phase completes, the dialog engine starts another Execution Phase. The `AskRegistered` agent will be popped from the dialog stack (this agent has completed since the `registered` concept is now updated). When `Login` will plan again, it will skip over `AskUserName` since the `user_name` concept is already available (the default precondition on request-agents is that the requested concept is not available). The next agent planned by `Login` will therefore be `GreetUser`, and the system responds: "Hello, John Doe … "

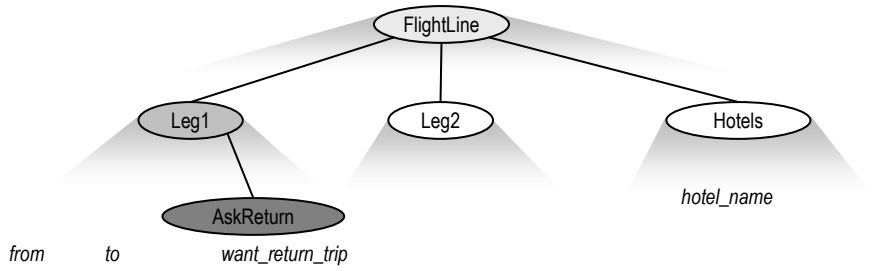
The expectation-agenda driven concept update mechanism provides a number of advantages: (1) it allows for the user to over-answer system questions, (2) in conjunction with the dialog stack, it provides support for mixed-initiative interaction, (3) it automatically performs context-based semantic disambiguation, and (4) it can provide a basis for dynamic state-specific language modeling. The first aspect was already illustrated in the example discussed above: the user not only answered the system question, but he also provided his name. In the next subsections we discuss in more detail the other three aspects listed above.

§ Expectation agenda and mixed-initiative interaction

The expectation agenda facilitates mixed-initiative conversation, since the system can integrate information from the user's response that does not necessarily pertain to the question in focus. In the previous example, we have illustrated a simple case in which the user over-answers a system question. More generally, in combination with the dialog stack, the expectation agenda allows the user to take initiative and shift the focus of the conversation.

We illustrate the focus-shift mechanism with an example from a spoken dialog system that operates in the air travel planning domain – see Figure 28. At turn n, the system question is "Will you be returning from San Francisco?", corresponding to the `/FlightLine/Leg1/AskReturn` agent in the dialog task tree. At this point, instead of responding to the system question, the user decides to ask about a booking a particular hotel in San Francisco. The decoded input matches the expectation for `[HotelName]`, on the last level in the agenda, and the `hotel_name` concept is updated accordingly. Once this Input Phase completes, the system continues with an Execution Phase. During focus claims analysis, the `/FlightLine/Hotels` agent claims focus, since this agent has a trigger condition that the `hotel_name` concept is updated: `TRIGGERED_BY(UPDATED(hotel_name))`. As a consequence, the dialog engine places this agent on top of the dialog stack – see the stack at time n+1 in Figure 28. As the dialog engine continues execution, the conversation continues from the `Hotels` dialog agency. Once the hotels sub-dialog completes, the `Hotels` agency is popped from the execution stack and we're back in the previous context, on the `AskReturn` question.

The system author can control the amount of initiative given to the user at every point in the dialog by controlling which expectations on the agenda are active and which expectations are closed (expectations that are closed will not bind). By default, the expectations defined by a request- or an expect-agent are active only when the focus of the conversation is under the same main topic as the



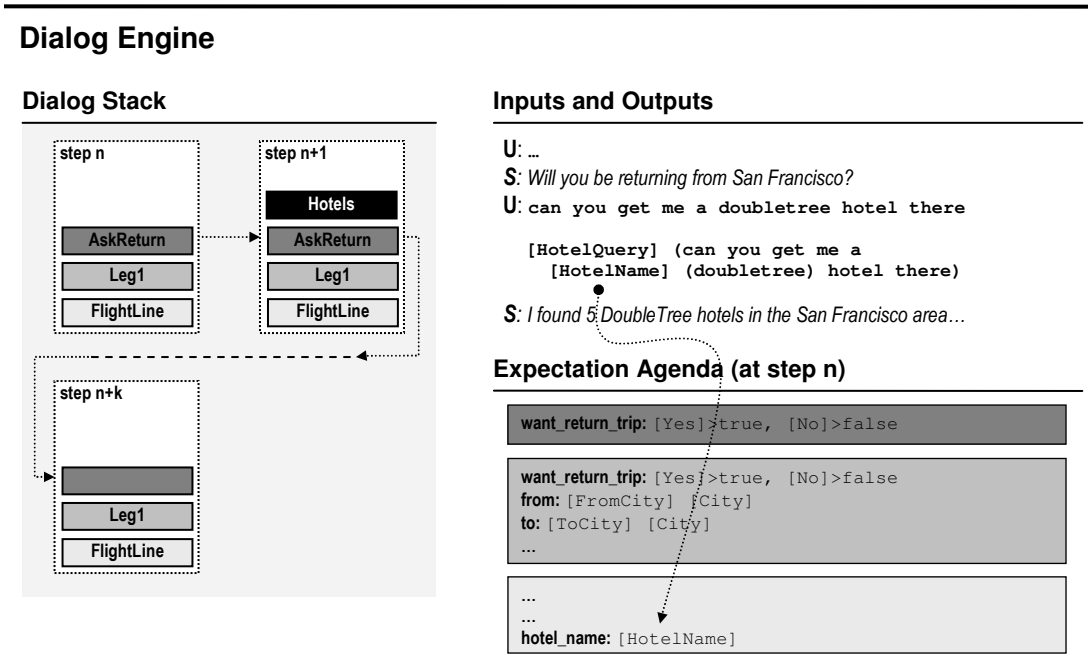


Figure 28. Focus-shift in a mixed-initiative conversation

agent that defines the expectation. For instance, if our `Hotels` agent was defined as a **main topic** (using the `IS_MAIN_TOPIC` directive), then the `[HotelName]` expectation would be closed at step n in Figure 28, and the `hotel_name` concept would not be updated. System authors can therefore control which expectations are active and which are closed by defining the main topics in the tree.

A finer-grained level of control can be achieved through **expectation scope operators**, which can be used to alter this default behavior:

- the `!` operator; when this operator is used while defining an expectation (e.g. `![Yes]>true`), the expectation will be active only when the agent that defines the expectation is actually in focus.
- the `*` operator; when this operator is used, the expectation is always open.
- the `@(<agent_name>;<agent_name>; …)` operator; the expectation is open only when the focus of the conversation is under one of agents in the specified list. For instance, if we wanted to allow the `hotel_name` concept to bind only while the conversation is on the first leg of the trip, but not the second leg of the trip, the expectation could be defined as: `@(/FlightInfo/Leg1;/FlightInfo/Hotels)[HotelName]`

The `EXPECT_WHEN` macro provides yet another degree of control over when expectations are active and when they are closed. This macro can be used on request- and expect-agents to define a Boolean condition that describes at which times the expectation should be open. System authors can

take into account any (state) information available to the dialog manager in order to control the opening and closing of expectations.

So far, we have discussed how system authors can control the amount of initiative the user is allowed to take at any point in the dialog. Note that the dialog engine could also automatically control the amount of initiative by limiting how deep in the agenda bindings are allowed. For instance, allowing bindings only in the first level in the agenda corresponds to a system-initiative situation, where the user is allow to respond only to the question in focus. While this type of behavior has not yet been implemented in the RavenClaw dialog engine, it is easy to envision a system where, perhaps depending on how well the dialog is progressing, the dialog engine automatically adjusts the level of initiative permitted to the user.

§ Expectation agenda and context-based semantic disambiguation

Another advantage provided by the expectation agenda is automatic resolution of some semantic ambiguities based on context. This feature appears as a side-effect of the top-down traversal of the agenda during the concept binding phase.

Consider the example from Figure 29, again drawn from a fictitious system operating in the air travel domain. The focus of the conversation is on the `AskFrom` request-agent, which is in charge of obtaining the departure city for the first leg of the trip. This agent declares two expectations for the `from_city` concept: `[FromCity]`, which captures constructs like for instance "I'd like to leave from San Francisco", and `[City]` which captures city names spoken in isolation, for instance "San Francisco". At the same time, the `AskTo` request agent in the `Arrival` subtree also declares the expectations `[ToCity]` and `[City]` in order to capture the arrival city (in the `to_city` concept). The user responds to the system question with a simple city name, which is semantically decoded as `[City]`. A semantic ambiguity arises: should this city bind to the `from_city` concept, or to the `to_city` concept? The ambiguity is automatically resolved given the top-down traversal of the agenda in the concept binding phase. In this case, the input updates the `from_city` concept, since this appears on the higher (in this case first) level in the agenda.

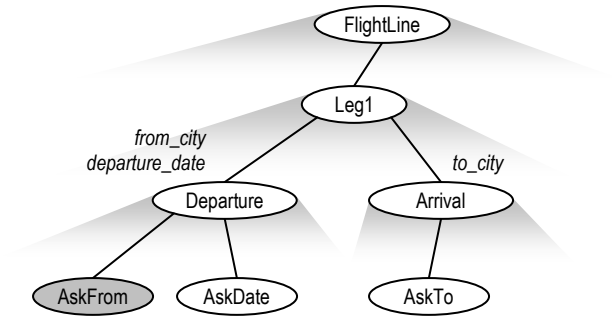
The expectation agenda therefore automatically implements an ambiguity resolution heuristic: if an input could be used to update more than one concept, always update the concept that is closest to the current context (higher in the agenda).

§ Expectation agenda and dynamic state-specific language modeling

The expectation agenda can also support dynamic, context-specific language modeling. At each turn in the dialog, the expectation agenda captures what the system expects to hear from the user, at the semantic level. This information could be used to dynamically construct a context-specific recognition language-model by interpolating a large number of smaller, fixed language models. For instance, considering the example from Figure 29, the system could create a state-specific language model by interpolating models `[Yes]`, `[No]`, `[FromCity]`, `[ToCity]`, `[City]`, etc. The level-based organization of the expectation agenda could also provide additional information about the likelihood of different user responses. The weights in the interpolation could be assigned based on the depth of the corresponding item in the expectation agenda. An advantage of this type of interpolated language models is that they can take into account the current context and dialog history in a more accurate fashion than a simple state-specific language model would (the expectation agenda might contain different grammar slots when a certain agent is in focus, depending on the dialog history). Secondly, the approach would not require the models to be retrained after each change to the dialog task structure. While this type of context-sensitive language-modeling approach has not yet been implemented in the RavenClaw dialog engine, Gruenstein and co-authors [43] have shown considerable reductions in word-error-rate under certain circumstances with a similar technique.

3.2.4 Task-independent conversational strategies

A characteristic that greatly influences the usability and ultimately the success of spoken dialog systems is their ability to engage in a rich set of conversational strategies. Apart from handling the actual



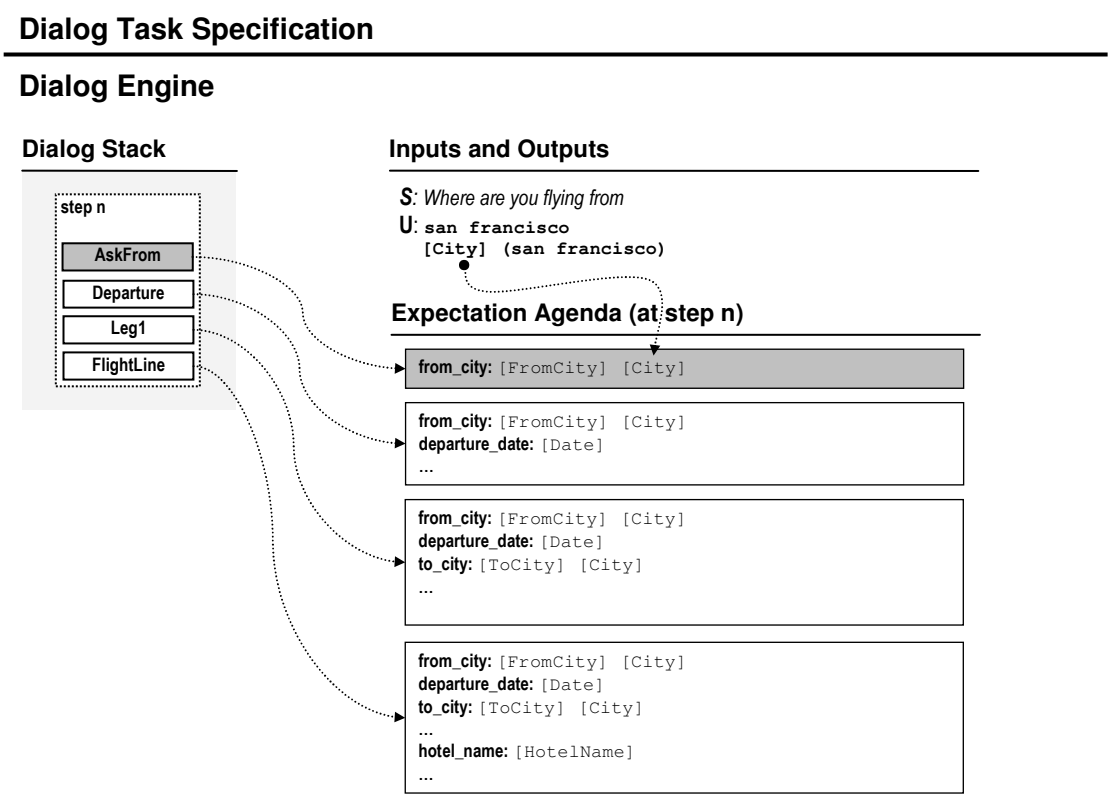


Figure 29. Context-based semantic disambiguation in the expectation agenda

dialog task, a dialog manager must be able to engage in error handling behaviors (e.g. explicit and implicit confirmation, disambiguation, asking the user to repeat, etc.), turn-taking and timing behaviors, as well as other generic dialog mechanisms, like the ability to handle requests for help, for repeating the last utterance, suspending and resuming the conversation, starting over, re-establishing the context, etc.

The RavenClaw dialog engine provides automatic support for a wide array of such conversational strategies. Internally, the strategies are implemented as library dialog agencies, using the same dialog task specification formalism as the domain-specific task tree. The system author simply specifies which strategies the dialog engine should use, and parameterizes them accordingly. The responsibility for invoking these strategies at the right time is delegated to the RavenClaw dialog engine. Additionally, developers can write new task-independent conversational strategies, encapsulate them as library agents, and make them available for use in other RavenClaw-based dialog systems. The current architecture promotes reusability, and ensures consistency in behaviors both within and across systems.

The task-independent conversational strategies currently available in the RavenClaw dialog

management framework fall into two categories: **(1) generic dialog mechanisms** (e.g. help, repeat, suspend, establish context, start over, etc.), and **(2) error recovery strategies** (e.g. explicit and implicit confirmation, asking the user to repeat, asking the user to rephrase, etc.) In the next two subsections we list and briefly discuss the strategies currently available in each class.

3.2.4.1 Generic dialog mechanisms

To date, six generic dialog mechanisms have been implemented and are available as library agencies in the RavenClaw dialog management framework; these mechanisms provide automatic support for servicing a number of domain-independent conversational requests like *"Help!"*, *"Repeat!"*, *"Where are we?"*, *"Start-over!"*, etc. The corresponding grammars for these requests are predefined in a common grammar library; however, system authors can further extend or modify these grammars. Similarly, a number of these strategies use predefined natural language generation templates (e.g. "Are you sure you would like to start over?") System authors can overwrite these templates, and customize them for a particular application.

We now briefly describe the six generic dialog mechanisms currently available in the RavenClaw dialog management framework.

§ Help

The RavenClaw dialog management framework currently supports five types of help requests: establish context, what-can-I-say, full-help, interaction-tips, and system-capabilities.

The `Help.EstablishContext` strategy provides information about the current system state in response to user requests like *"Where are we?"* This strategy issues the `establish_context` prompt for the request agent that was in focus (i.e. on top of the dialog stack) at the time the help request occurred. The content of the `establish_context` prompts is provided by the system author. Generally, these prompts provide information about the system state, for instance "I am currently trying to collect enough information to make a room reservation for you. So far, I know you need a room on Friday starting at 10 a.m."

The `Help.WhatCanISay` strategy provides examples for what the user could say at this point in the dialog, in response to requests like *"What can I say?"* The strategy issues the `what_can_i_say` prompt for the request agent that was in focus at the time the help request occurred. These prompts are also provided by the system author, and they contain information such as "For instance, you could say something like 'until 2 p.m.', or 'for two hours' ".

The `Help.FullHelp` strategy provides a full help message to the user, in response to a simple help request - *"Help!"* This strategy issues in sequence the `establish_context`, `explain_more`, and `what_can_i_say` prompts for the request agent previously in focus. For instance, a full help prompt could be: "I am currently trying to collect enough information to make a room reservation for you. So far, I know you need a room on Friday starting at 10 a.m. Right now, I need you to tell me until what time you need this room. For instance, you can say something like 'until 2 p.m.' or 'for two hours' ". The `Help.FullHelp` strategy can also be configured to provide a shorter help message by omitting the `establish_context` prompt.

The `Help.InteractionTips` strategy provides a generic help message containing guidelines for how to best interact with the system. Systems authors customize the corresponding language generation template. A typical example is: "Here are some tips for a smooth interaction. Please speak clearly and naturally. Do not speak too quickly or too slow. You can interrupt the system at any time by saying anything you wish. If you need to make a correction, just restate the new information. For example, if you'd like a room in the afternoon instead of morning, you can simply say, 'I'd like a room in the afternoon'. To get help at any time, please say 'help'. To hear what your options are at any point, say, 'what can I say'. To hear a summary of the system's state, say, 'where are we' ".

Finally, the `Help.SystemCapabilities` strategy provides a generic message about the system's capabilities, in response to user requests like *"What can you do?"* Again, system authors have to customize the corresponding language generation template. For instance, in the RoomLine system,

this prompt is: "I can assist you to make conference room reservations in S C S. I have information about 13 rooms throughout Newell Simon and Wean Hall. For instance, if you want to reserve a room for next Wednesday, you can say: 'I want a room for next Wednesday from ten until noon.' I also have information about the equipment available in each of the rooms, like overhead projectors, whiteboards, room size, and networking. You could say for instance, I want a room with a projector and a whiteboard for tomorrow morning. If you want more hints about how to interact with the system, say, 'interaction tips' Now, moving back to where we were …".

§ Quit

The `Quit` strategy allows users to terminate a conversation at any point by saying something like *"Quit!"* or *"Good-bye"*. The strategy asks a verification question "Are you sure you would like to terminate this session?" Upon confirmation, the dialog is terminated.

§ Repeat

The `Repeat` strategy implements support for repeating the previous system prompt, for instance if the user says *"Repeat!"* or *"What?"*

§ StartOver

The `StartOver` strategy allows users to restarting the conversation from the beginning. After the system confirms the user's intention to restart, the dialog task tree is reset to the initial configuration (e.g. all agents are reset, all concepts are cleared.) In addition, the system author can also specify a customized start-over routine. This is useful in situations when the system must retain some information on a start-over, rather than start from scratch.

§ Suspend

The `Suspend` strategy implements support for temporarily suspending the conversation, on a user request like *"Suspend!"* or *"Hold on a minute."* The system informs the user that the conversation is temporarily suspended, and the user can restart by saying *"Resume conversation!"* Once the resume command is given, the strategy first issues the `establish_context` prompt on the agent that was previously in focus, and then completes; the conversation then continues from where it was left over.

§ Timeout

There are two `Timeout` strategies available in the RavenClaw dialog management framework. Both handle situations in which no user response is received within a specified timeout interval since the end of the last system prompt. Timeout intervals can be defined globally, and overwritten locally through the `TIMEOUT_PERIOD` macro on individual dialog agents. If the timeout period elapses and no response is received from the user, the RavenClaw dialog engine creates an internal `[TIMEOUT]` event, which triggers the `Timeout` strategy.

The first timeout strategy, `Timeout.Terminate`, attempts to reestablish the channel by first issuing the `timeout` prompt corresponding to the previously focused agent. If that fails, the strategy tries again to reestablish the channel by asking "Are you still there?" If the system still does not receive an answer, this strategy will terminate the conversation by issuing a prompt like "I assume you are no longer there. I will hang up now."

The second timeout strategy, `Timeout.Suspend`, suspends the conversation when a timeout occurs. The user can resume the conversation by saying something like *"Resume"* or *"Let's restart"*. System authors specify which one of these two timeout strategies the dialog manager should use.

3.2.4.2 Error recovery strategies

The error recovery strategies in the RavenClaw dialog management framework fall into two categories: (1) strategies for handling potential misunderstandings, and (2) strategies for handling non-understandings. Just like the other task-independent conversational strategies, the error recovery strategies are implemented as library dialog agencies. The dialog engine monitors the conversation and, during the error handling phase it decides whether or not it needs to engage in any error han-

Error Handling Strategy	Example
Strategies for handling misunderstandings	
Explicit confirmation	Did you say you wanted a room starting at 10 a.m.?
Implicit confirmation	starting at 10 a.m. … until what time?
Strategies for handling non-understandings [suppose a non-understanding happens after the system asks: "Would you like a small room or a large one?"]	
Notify that a non-understanding happened	Sorry, I didn't catch that …
Ask user to repeat	Can you please repeat that?
Ask user to rephrase	Can you please rephrase that?
Repeat prompt	Would you like a small room or a large one?
Give a you-can-say help message	For instance, you could say something like "I want a small room", or "I want a large room"
Give an explain-more help message	Right now I need you to tell me if you would prefer a small room or a large room.
Give a full help message	I found seven rooms available Friday from 10 to 12. Right now I need you to tell me if you would prefer a small room or a large room. For instance, you could say something like "I want a small room", or "I want a large room"
Give tips about how to best interact with the system	Okay, I know this conversation isn't going well. There are things you can try to help me understand you better. Speak clearly and naturally; don't speak too quickly or too slowly. Give short, concise answers. Calling from a quiet place helps. If you'd like to start from scratch, you can say 'start-over' at any time.
Ask for a short answer and repeat prompt	Please use shorter answers because I have trouble understanding long sentences… Would you like a small room or a large one?
Ask for a short answer and give a you-can-say help message	Please use shorter answers because I have trouble understanding long sentences… For instance, you could say something like "I want a small room", or "I want a large room"
Ask user to speak less loud and repeat prompt	I understand people best when they speak softer. Would you like a small room or a large one?
Fail the current request and move-on	Sorry, I didn't catch that. One choice would be Newell Simon 1507. This room can accommodate 50 people, and has a projector, a whiteboard and network access. Do you want a reservation for Newell Simon 1507?
Yield the turn	[…] (system remains silent, yielding the turn to the user)
Ask user if they'd like to start over	I'm sorry I'm still having trouble understanding you, and I might do better if we restarted. Would you like to start over?
Give up	I'm sorry but I'm having lots of trouble understanding you and I don't think I will be able to help you. Please call back during normal business hours.

Table 11. Task-independent error handling strategies in the RavenClaw dialog management framework

dling action. If an action is deemed necessary, an instance of the corresponding error recovery strategy is created on the fly, and dynamically added to the dialog stack. In the next chapter, we will describe the RavenClaw error handling architecture, and give a more detailed presentation of each of these strategies, and of the mechanisms used to invoke them. Here, we enumerate the strategies, and provide examples for each of them – see Table 11.

3.3 The Olympus dialog system infrastructure

In the previous section, we have provided an overview of the RavenClaw dialog management framework. We have seen that RavenClaw is a two-tier architecture that decouples the domain-specific aspects of the dialog control logic from the domain-independent dialog engine. To build a new dialog manager, system authors have to develop a dialog task specification, essentially a hierarchical plan for the interaction. The dialog engine then manages the dialog by executing this dialog task specification. However, in order to build a fully functioning spoken language interface, a number of other components besides a dialog manager are required: speech recognition, language understanding and generation, speech synthesis, etc.

In this section we give a quick overview of Olympus, a collection of freely available dialog system components (and corresponding control logic) that we have used in conjunction with RavenClaw to build and deploy spoken language interfaces. Then, in section 3.4, we will discuss a number of spoken dialog systems that have been developed within the RavenClaw/Olympus framework. These systems provide the experimental platform for the error handling research program discussed in the rest of this dissertation.

Olympus [12] is a dialog system infrastructure that, like RavenClaw, has its origins in the earlier CMU Communicator project [101]. At the high-level, Olympus consists of a series of components connected in a classical, pipeline architecture – see Figure 2. The audio signal for the user utterance is captured and passed through a speech recognition module that produces a recognition hypothesis (e.g. *"two p.m."*). The recognition hypothesis is then forwarded to a language understanding component that creates a corresponding semantic representation, e.g. `[time=2p.m.]`. Next, the RavenClaw-based dialog manager integrates this semantic input into the current discourse context, and produces the next system action in the form of a semantic output (e.g. `{request end_time}`). A language generation module produces the corresponding surface form, which is subsequently passed to a speech synthesis module and rendered as audio.

Figure 31 provides a more detailed account of a typical RavenClaw/Olympus based system. While the pipeline illustrated in Figure 2 captures the logical flow of information in the system, in practice the various system components do not communicate directly. Rather, they rely on a centralized message-passing infrastructure – Galaxy [111]. Each component is implemented as a separate process that connects to a centralized traffic router – the Galaxy hub. The messages are sent through the hub, which forwards them to the appropriate destination. The routing logic is described by means of a configuration script.

For recognition, Olympus uses the Sphinx decoding engine [58]. A recognition server component captures the audio stream (typically from the sound-card), forwards it to a set of parallel recognition engines, and collects the corresponding recognition results. All the top-level recognition hypotheses (one from each engine) are then forwarded to the language understanding component. Currently, Sphinx-II (semi-continuous HMM recognition) and Sphinx-III (continuous HMM recognition) engines are available and can be used in conjunction with the recognition server. The individual recognition engines can be configured to use a variety of off-the-shelf acoustic models, and either n-gram or grammar-based language models. State-specific as well as class-based language models are supported, and tools for constructing language and acoustic models from data are readily available.

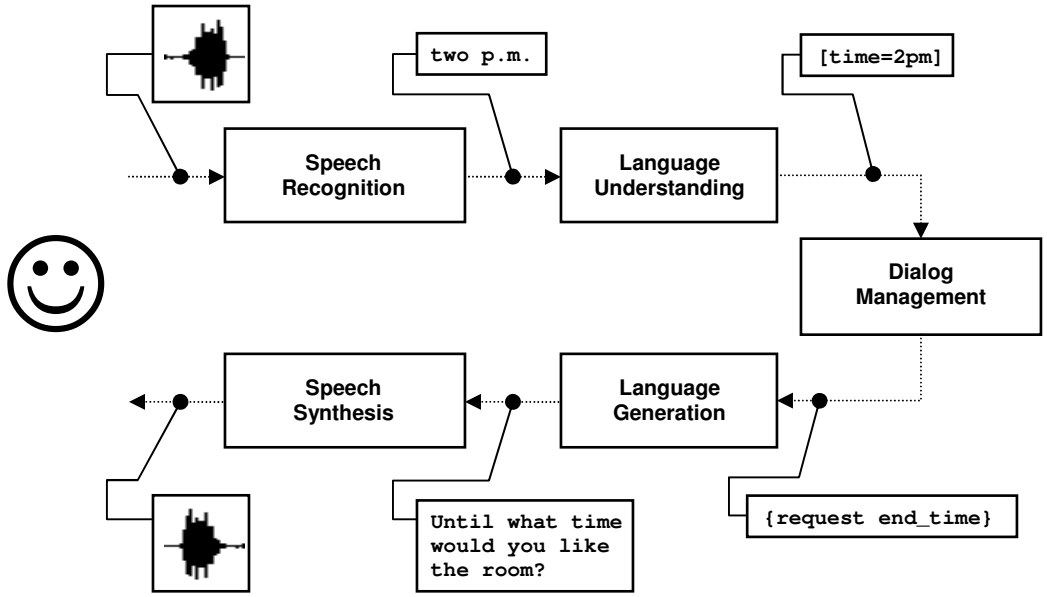


Figure 30. Olympus: a classical dialog system architecture

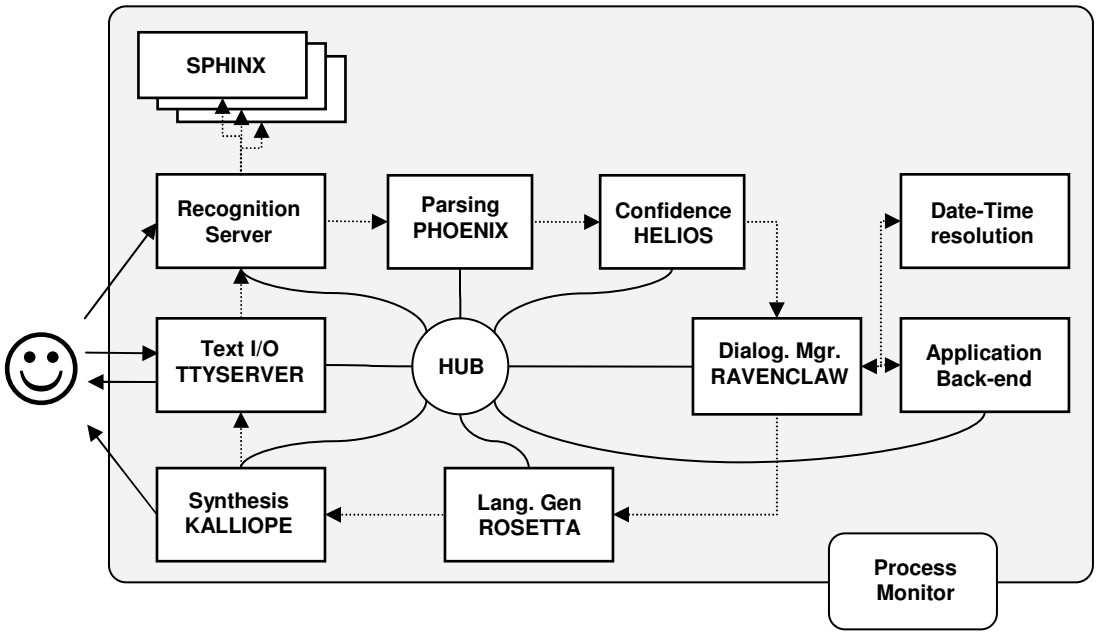


Figure 31. The Olympus/RavenClaw dialog system architecture: a more detailed view

Additionally, a DTMF (touch-tone) decoder is also available as a recognition engine. Most of the RavenClaw/Olympus systems described in the next section use two parallel Sphinx-II recognizers: one configured with acoustic models trained using male speech and the other configured with acoustic models trained using female speech. Other parallel decoder configurations can also be created and used.

Language understanding is implemented via Phoenix, a robust semantic parser [131]. Phoenix uses a semantic hand-written grammar to parse the incoming set of recognition hypotheses (one or more parses can be generated for each hypothesis). The semantic grammar is constructed by concatenating a set of reusable grammar rules that capture domain-independent constructs like `[Yes]`, `[No]`, `[Help]`, `[Repeat]`, `[Number]`, etc., with a set of domain-specific grammar rules authored by the system developer. For each recognition hypothesis the output of the parser consists of a sequence of slots containing the concepts extracted from the utterance.

From Phoenix, the set of parsed hypotheses is passed to Helios, the confidence annotation component. Helios uses features from different knowledge sources in the system (e.g. recognition, understanding, dialog, etc.) to compute a confidence score for each parsed hypothesis. This score reflects the probability of correct understanding, i.e. how much the system trusts that the current semantic interpretation corresponds to the user's expressed intent. The hypothesis with the highest confidence score is then forwarded to the dialog manager.

The next component in the chain is the RavenClaw-based dialog manager. The dialog manager integrates the semantic input in the current discourse context, and decides which action the system should engage in next. In the process, the dialog manager may consult / exchange information with a number of other domain-specific agents, such as an application-specific back-end.

The semantic output from the dialog manager is then processed by the Rosetta language generation component, which creates the corresponding surface form. Rosetta supports template-based language generation. Like the grammar, the set of language generation templates is assembled by concatenating a set of predefined, domain-independent templates, with a set of manually authored task-specific templates.

Finally, the prompts are synthesized by the Kalliope speech synthesis module. Kalliope can be configured to use a variety of speech synthesis engines: Festival [8], which is an open-source speech synthesis system, as well as Theta [24] and Swift [24], which are commercial solutions. Kal-

liope supports both open-domain (e.g. diphone) and limited-domain (e.g. unit selection) voices. The SSML markup language is also supported.

The various components briefly described above form the core of the Olympus dialog system infrastructure. Additional components have been added throughout the development of various systems, and, given the modularity of the Olympus architecture, they can be easily reused. Here are some examples. A text input-and-output server (TTYServer) provides support for text-based systems, and can also be very useful for debugging purposes. More recently a GoogleTalk agent was implemented in the Olympus framework to support interactions via the popular internet messaging system. A Skype speech client component is also available. A Process Monitor component is used to start-up and to monitor all the other components in an Olympus system. Additionally, the Process Monitor can automatically restart components that crash, and send notification emails to a system administrator. Finally, a variety of logging and data processing and analysis tools are also available as part of the Olympus distribution.

3.4 RavenClaw-based systems

Evaluation of spoken language interfaces is a difficult task, and has received significant amounts of attention from the research community [48, 55, 127-129]. Evaluating a dialog management framework poses even harder challenges. To our knowledge, no such objective assessments have been performed to date. Characteristics such as ease-of-use, portability, domain-independence, scalability, robustness, etc. are very hard to capture in a quantitative manner. In a sense, perhaps the problem is ill-posed. Comparing dialog management frameworks is like comparing programming languages: while arguments can be made about various strengths and weaknesses of various programming languages, no clear order or measure of absolute performance can be established. Certain programming languages are more appropriate for certain tasks than others. Even evaluating the suitability of a programming language (or dialog management framework) for a given task is difficult, since many applications can be recast into a form that is tractable in a particular approach.

As a first step towards a more rigorous evaluation of the RavenClaw dialog management framework, we decided to use this framework to build a number of spoken dialog systems spanning different domains and interaction types. In the process, we monitored various aspects of the development process and noted the degree of accommodation required by RavenClaw.

To date, about a dozen such systems have been developed using the RavenClaw dialog management framework and the larger Olympus infrastructure. Some of these systems have been deployed successfully into day-to-day use. Table 12 provides a quick summary of 8 of these systems. As the table shows, these systems operate in structurally different domains and were constructed by development teams with different degrees of experience building spoken language interfaces.

In the following subsections, we briefly describe each of these systems. For each system, we present more details about the domain, the interaction style, as well as various system characteristics such as vocabulary and grammar size, etc. Whenever available, we present performance statistics and discuss issues encountered throughout development and deployment stages. In addition, we also describe research issues that have been or are currently investigated in the context of these systems. I the last subsection, 3.4.9, we summarize our overall experience in using the RavenClaw dialog management framework in these application domains.

3.4.1 RoomLine

RoomLine is a telephone-based mixed-initiative spoken dialog system that provides access to conference room schedule information and allows users to make conference room reservations. The system has access to live information about the schedules of 13 conference rooms in 2 buildings on the CMU campus: Wean Hall and Newell Simon Hall. Additionally, the system has information about the various characteristics of these rooms such as location, size, network access, whiteboards, and audio-visual equipment. To perform a room reservation, the system finds the list of rooms that sat-

System name	Domain / Description	Interaction type	Developers
RoomLine	telephone-based system that provides support for conference room reservation and scheduling within the School of Computer Science at CMU.	information access (mixed initiative)	Bohus D.
Let's Go! Public [87-89]	telephone-based system that provides access to bus route and scheduling information in the greater Pittsburgh area	information access (system initiative)	Raux A., Bohus D., Langner B., Black A., Eskenazi M.
LARRI [13]	multi-modal system that provides assistance to F/A-18 aircraft personnel during the execution of maintenance tasks	multi-modal task guidance and procedure browsing	Bohus D., Sun Y., Patel K., Chotimongkol A.
Intelligent Procedure Assistant [1, 2]	early prototype for a multi-modal system aimed at providing guidance and support to the astronauts on the International Space Station during the execution of procedural tasks and checklists	multi-modal task guidance and procedure browsing	RIALIST group/NASA Ames, Aist G., Bohus D.
TeamTalk [47]	multi-participant spoken language command-and-control interface for a team of robots operating in the treasure-hunt domain	multi-participant command-and-control	Harris T.K., Banerjee S., Sison J., Kishore S.P., Bodine K., Bohus D.
VERA	telephone-based taskable agent that can be instructed to deliver messages to a third party and make wake-up calls.	message-passing	Bardak U., Judy S., Pedro V., Blum T., Ko J., Miyata R.
Madeleine	text-based dialog system for medical diagnosis	diagnosis	Bohus D.
ConQuest [11]	telephone-based spoken dialog system that provides conference schedule information (deployed during Interspeech-2006)	information access (mixed-initiative)	Bohus D., Kumar, R., Krishna G., Keri V., Grau S., Tomko S., Raux A.

Table 12. RavenClaw-based spoken dialog systems

isfy an initial set of user-specified constraints. Next, RoomLine presents this information to the user, and engages in a follow-up negotiation dialog to identify which room best matches the user's needs. Once the desired room is identified, registered users can authenticate using a 4-digit touch-tone PIN, and the system performs the reservation through the campus-wide CorporateTime calendar server. Following the interaction, the user also receives a confirmation email for the room reservation. A sample conversation with the system is presented in Appendix A.

RoomLine engages in a slot-filling type interaction in the initial phase of the dialog, and a negotiation-type interaction in the second part of the dialog, where users can take the initiative and navigate the solution space by specifying additional constraints, or relaxing the existing ones. The plan-based RavenClaw dialog management framework successfully supports both interaction types.

The current (as of December 2006) version of the RoomLine system uses two parallel Sphinx-II engines for recognition equipped with gender specific acoustic models. The vocabulary size is 1092 words. The semantic grammar used for language-understanding purposes contains 45 top-level grammar slots. The system uses a confidence annotation model that leverages multiple knowledge sources and is trained with in-domain data according to the methodology described in Chapter 5. The dialog task specification contains 117 dialog agents and 28 concepts. The dialog manager is configured to use both explicit and implicit confirmations to recover from potential misunderstandings, and ten strategies to recovery from non-understandings. The system uses template-based language generation, and a Cepstral-based [24] open-domain unit-selection synthesizer. In a scenario-driven in-lab experiment (described later in Chapter 8), the system attained an average task-success rate of 75.1% (85.2% for native speakers and 44.1% for non-native speakers) at an average word-error-rate of 25.7% (19.7% for native speakers, and 39.8% for non-native speakers).

RoomLine was one of the first systems developed using the RavenClaw dialog management framework. The system has been publicly deployed in 2003, and has been available 24x7 to students

and faculty on campus ever since. Overall, RoomLine provides one of the main experimental platforms for the implementation and evaluation of the error handling research described in the rest of this dissertation. Data collected with this system provides the basis for our empirical error analysis presented in Chapter 2, our experiments in confidence annotation and belief updating described in Chapter 5 and Chapter 6, as well as our investigation of non-understanding recovery strategies described in Chapter 8.

3.4.2 Let's Go! (Public) Bus Information System

The Let's Go! Public Bus Information system [87-89] is a telephone-based spoken dialog system that provides access to bus route and schedule information. The system knows about 12 bus routes, and 1800 place names in the greater Pittsburgh area. In order to provide bus schedule information, the system tries to identify the user's departure and arrival stop, and the departure or arrival time. Once the results are provided, the user can ask for the next or previous bus on that route, or can restart the conversation from the beginning to get information for a different route. A sample interaction with this system is illustrated in Appendix A. Additional information is available on the system's web-site: `http://www.cmuletsgo.org`.

Let's Go! Public is therefore a slot-filling, information-access system. The conversation begins with an open-ended "How can I help you?" prompt, but continues with a set of focused questions in which the system asks in order for the departure place, arrival place and travel time. The user is allowed to over-answer any particular question, but all concept values are explicitly confirmed by the system before moving on. After the results are presented, the system offers a menu-style set of options for finding more information about the current selected route or restarting the conversation. No significant challenges were encountered in implementing this type of interaction using the RavenClaw dialog management framework.

Let's Go! Public uses three parallel decoding engines: two gender-specific Sphinx-II engine and a touch-tone recognition engine. The vocabulary size is 7911 words (including variants for 1800 place names on 12 bus routes.) The system uses state-specific, class-based tri-gram language models. The semantic grammar contains 29 top-level grammar slots. The confidence annotation model leverages multiple knowledge sources in the system and is trained with in-domain data according to the methodology described in Chapter 5. The dialog task specification contains 62 dialog agents and 17 concepts. The current version of this system uses a pessimistic policy for handling potential misunderstandings: the system explicitly confirms all concept values received from the user. While initially the system used 5 strategies for recovering from non-understandings, in February 2006 this set of strategies was refined and expanded [87]. The policy used to engage these strategies was learned online, using an approach we will later describe in section 8.4 from Chapter 8. Let's Go! Public uses template-based language generation and a Cepstral-based [24] open-domain unit-selection synthesizer.

The Let's Go! Public Bus Information system was developed as a follow-up, public version of the earlier Let's Go! Bus Information system. This first system, also based on the RavenClaw-Olympus architecture, was developed earlier at CMU as part of a research project focused on investigating methods for making spoken language interfaces more accessible to elderly and non-native speakers. In fall of 2004 the management of the Port Authority of Allegheny County (PAAC) called the Let's Go! experimental system and found that it could correspond to the user's needs. After a redesign stage aimed to increase robustness, the new version of the system, dubbed Let's Go! Public was open to the Pittsburgh population on March 4th, 2005. The system is connected to the PAAC customer service line during non-business hours, when no operators are available to answer the calls (i.e. 7 p.m. to 6 a.m. on weekdays, and 6 p.m. to 8 a.m. on weekends and national holidays.) Since it deployment, the system has serviced on average about 40-50 calls per night, for a total of over 30,000 calls.

The constant and relatively large traffic to the system, together with the real-world nature of the application (the calls come from users with real needs) make it an excellent platform for research.

To date, a number of studies and investigations have been performed using this system. With respect to the work described in this dissertation, the Let's Go! Public system provided the experimental platform for investigating an online supervised approach for learning non-understanding recovery strategies described in Chapter 8. In other work [89], Raux et al. have performed an analysis of user responses to various non-understanding recovery strategies in this system. In addition, in [87], Raux et al have investigated the impact of four different factors (recognition accuracy, turn-taking errors, non-understanding recovery strategies, initial prompt) on overall dialog performance. More generally, the Let's Go! Public system currently provides the experimental test-bed for the development and evaluation of RavenClaw/Olympus-II, the next version of the RavenClaw/Olympus infrastructure which addresses research issues related to timing and turn-taking in conversation [86]. Future plans with the Let's Go! Public system also include opening up the system for experimentation to other researchers in the spoken language interaction community.

3.4.3 LARRI

LARRI [13], or the Language Based Retrieval of Repair Information system is a multi-modal system for support of maintenance and repair activities for F/A-18 aircraft mechanics. The system implements a level 4/5 IETM (Interactive Electronic Technical Manual), that is, semantically annotated documentation. LARRI integrates a graphical user interfaces for easy visualization of dense technical information (e.g. instructions, schematics, video-streams, etc.) with a spoken dialog system that facilitates information access and offers assistance throughout the execution of procedural tasks.

A sample interaction with LARRI is presented in Appendix A. After the user logs into the system, LARRI retrieves the user's profile from a backend agent, and allows the user to view their current schedule and to select a task to be executed. The typical maintenance task consists of a sequence of steps, which contain instructions, optionally followed by verification questions in which the possible outcomes of each step are discussed. Basic steps can also include animations or short video sequences that can be accessed by the user through the GUI or through spoken commands. By default, LARRI guides the user through the procedure, in a step-by-step fashion. At the same time, the user can take the initiative and perform random access to the documentation/task, either by accessing the GUI or by simple spoken language commands such as *"go to step 15"* or *"show me the figure"*. Once the maintenance task is completed, the system provides a brief summary of the activity, updates the information on the back-end side, and moves to the next scheduled task.

In contrast to RoomLine and Let's Go! Public systems, which operate in information-access domains, LARRI provides assistance and guidance throughout the execution of a procedural task. The hierarchical representation of the dialog task used in RavenClaw is also very well suited in this domain, as it maps directly onto the structure of the actual tasks to be performed by the user (with sub-tasks, steps, sub-steps, etc). Moreover, since the procedural tasks are extracted on-the-fly from a task repository, the framework's ability to dynamically generate/expand the dialog task specification at runtime plays a very important role. Another important difference is that LARRI is a multi-modal application that integrates a graphical user interface for easy visualization of dense technical information (e.g. instructions, video-sequences, animations, etc.) with a spoken language interface that facilitates information access and offers task guidance in this environment. The graphical interface is accessible via a translucent head-worn display connected to a wearable client computer. A rotary mouse (dial) provides direct access to the GUI elements. The GUI is connected via the Galaxy hub [111] to the rest of the system: the rotary mouse events are rendered as semantic inputs and are sent to Helios which performs the multimodal integration and forwards the corresponding messages to the dialog manager.

Speech recognition is performed via the Sphinx-II decoder, using semi-continuous HMMs. Acoustic models were trained using the Wall Street Journal (WSJ-0) corpus, and a class-based, trigram language model was constructed and used for recognition. The current vocabulary contains 408 words, and the semantic grammar contains 61 top-level grammar slots. The system uses a simple confidence annotation model that relies on a heuristic based on a goodness-of-parse score. The dia-

log task specification is constructed dynamically, based on the task that the user is currently executing. The number of agents and concepts in the task can therefore grow from a minimum of 61 to several hundreds, depending on length of the current task. In practice, the system scaled up gracefully. LARRI uses template-based language generation, and a Festival-based [8] open-domain unit-selection synthesizer.

Although this system was never deployed into day-to-day use, we did evaluate LARRI on two separate occasions. The evaluations were performed on a military base, with the participation of trained Navy personnel, and the focus was on understanding the experience of the mechanics. While users commented favorably on the language-based interface, a closer analysis of the sessions and feedback revealed a number of issues. They included the need for better feedback on the system's state; a more flexible (controllable) balance between the spoken and graphical outputs, as well as improvements to the GUI design.

An interesting research question raised by the development of the LARRI system was: how can we automatically prepare the necessary resources (e.g. language models, grammars, generation templates, etc.) for a spoken language interface for a technical manual? (assume we are given the documentation in a PDF format.) While this is not a critical problem for small closed-domain applications, it needs to be addressed for large and constantly mutating domains such as aircraft maintenance.

3.4.4 Intelligent Procedure Assistant

The Intelligent Procedure Assistant (IPA) [1, 2] was an early prototype of a multimodal spoken dialog system meant to assist astronauts on the International Space Station during the execution of checklists and various procedural tasks. This system is very similar in domain, interaction-style and challenges to the LARRI system (F-18 maintenance task) described in the previous subsection. Like LARRI, the IPA system allows users to browse through a procedure, either one-line-at-a-time or in larger steps. The system also allows users to request images and diagrams associated with various steps in the checklist, record voice notes and associate them with various steps, and control the audio volume.

In contrast to the other RavenClaw-based systems described in this section, the IPA system was not constructed using the Olympus infrastructure and components. This system was developed in collaboration with the RIACS group at NASA Ames and used a series of other spoken language processing components connected using the Open Agent Architecture [72]. Speech recognition was accomplished using a Nuance 8 recognizer with a context-free language model constructed from a unification grammar and then compiled into a recognition model [92]. The top recognition hypothesis was parsed using the Gemini parser [33] and finally rendered as predicate-argument structures such as `volume(up)`. The system used a RavenClaw-based dialog manager. A few changes were required in the RavenClaw dialog management framework to support inputs in predicate-argument form and a connection with the Open Agent Architecture. Given the modularity of the RavenClaw architecture, these changes were easy to make: an additional dialog interface class (supporting OAA) and input processing class (supporting predicate-argument inputs) were defined. The visual display was implemented as an HTML document rendered in a regular web browser (the current step and sub-step was highlighted accordingly at each point in the dialog). Speech synthesis was handled by AT&T's speech synthesizer, equipped with a customized pronunciation dictionary.

Overall, no significant challenges were encountered in implementing the IPA dialog manager using RavenClaw, or in using the RavenClaw dialog management framework in a different environment, i.e. OAA communication infrastructure, and different input and output representations. Although IPA was only an early demonstration prototype, the system was well received. Its successor, Clarissa [93], was the first spoken language interface tested in outer space.

3.4.5 TeamTalk

TeamTalk [47] is a multi-participant spoken language interface that facilitates communication between a human operator and a team of robots. The system operates in a multi-robot-assisted treasure-hunt domain. The human operator is tasked to search a space for objects of interest and to bring those objects to a specified location. To achieve this task, the human operator directs the robots in a search operation. For instance, in one experiment [47], users were required to navigate (only by using the speech channel) two robots through corridors towards a certain location in a building.

The domain is command-and-control in flavor, but poses a number of additional challenges due to the multi-participant nature of the conversation. On the input side, the robots need to be able to identify who the addressee of any given user utterance is. On the output side, the robots need to address the problem of channel contention (i.e. multiple participants speaking over each other). For the interested reader, details about the current solutions to these problems are discussed in [47]. More generally, TeamTalk constitutes an excellent research platform for multi-agent dialog dynamics. Apart from the multi-participant aspects, some of the other current research goals in this project are: (1) understanding the skills needed for communication in a human-robot team, (2) developing languages for robot navigation in novel environments and (3) understand how novel objects, locations and tasks come to be described in language.

The RavenClaw/Olympus framework was relatively easily adapted to the demands of this domain. In the current architecture, each robot uses its own RavenClaw-based dialog manager, but all robots share the other Olympus components: Sphinx-II based speech recognition, Phoenix-based language understanding, Rosetta-based language generation and Festival-based speech synthesis (each robot uses a different voice.) TeamTalk can interface with real robots, including the Pioneer P2DX and the Segway RMP. In addition, it can interface with virtual robots within the high-fidelity USARSim [4] simulation environment. The processed user inputs are sent to all dialog managers (robots) in the system; each dialog managers decides based on a simple algorithm [47] whether or not the current input is addressed to it. If so, an action is taken; otherwise the input is ignored (it will be processed and responded to by another robot.) A few small changes were required in the RavenClaw/Olympus architecture to support multiple dialog managers. For instance, the RavenClaw output messages were augmented with information about the identity of the dialog manager that generated them; this information was later used by the synthesis component to decide on the voice to use.

3.4.6 VERA

VERA (Voice Enabled Reminder Assistant) is a taskable agent that can be instructed to deliver messages to a third party, make wake-up calls, etc. The system consists of two different telephone-based spoken language interfaces, each constructed using the RavenClaw/Olympus infrastructure. The first interface – Vera In – handles incoming requests and places the desired reminders and messages in a database. The messages are saved as full audio files, and subsequently delivered as such (no speech recognition is performed on them.) The second system – Vera Out – continuously polls the same database and initiates calls to deliver the messages. Apart from the messages to be delivered, the database also contains contact information with various phone numbers / locations for each registered user and potential message recipient. The system cycles through these numbers, according to a predefined preference ordering. In each call the system starts by greeting the person that picked up the phone, and then tries to identify whether or not they are the desired message recipient. If the desired recipient is reached, the corresponding message is delivered. Otherwise the system continues to try the other numbers until the intended recipient is reached.

VERA is interesting in that, in contrast to typical spoken language interfaces, it does not simply receive calls and provide information. It also initiates calls. The dialog in which the system attempts to identify the person at the end of the line (who might be an answering machine or who might not expect to be called by an automated system) poses a number of interesting research challenges.

The system was developed as part of a class project, by a team of six students who had no

prior experience with RavenClaw or Olympus. Modulo an initial lack of documentation, no major problems were encountered in the development of this system. In terms of the actual infrastructure, the VERA In and VERA Out systems are very similar to the other RavenClaw/Olympus systems described so far. One notable difference is that a VoIP Skype-connection component was written for this system and integrated with the rest of the Olympus infrastructure.

3.4.7 Madeleine

Madeleine is a text-only dialog system implemented in response to a workshop challenge problem launched by MITRE in Fall of 2003. The challenge problem involved the rapid development of a spoken dialog system for medical diagnosis, where the application back-end was provided by MITRE Corporation. The back-end consisted of a set of disease diagnostic trees. At each point in the dialog, the system could collect more information about symptoms from the user or perform various tests until a unique diagnosis was reached.

Like LARRI, the Madeleine system dynamically generates the dialog task specification. The set of possible symptoms are loaded from the back-end, and the corresponding dialog task structures for talking about or performing the associated tests are created on the fly. Furthermore, while the default execution policy in the RavenClaw dialog management framework was an in-order traversal of the task tree (see subsection 3.2.2.2), in this case a customized execution policy was used. The system made relies of the decision trees in the application backend to determine which symptom should be investigated next, i.e. which subagent should be planned for execution by the root diagnosis agency.

Throughout the development of this system we performed an informal wall-clock analysis of the amount of time required to build each of the system components. The total time has 21 hours and 15 person-minutes. Note that the developer of this system – the author of this dissertation – had expert knowledge of the RavenClaw/Olympus framework. Figure 32 illustrates the break-down of this time into various components. It is interesting to notice that no single component dominated the development time; rather, a number of hours were dedicated to each component. The interface design stage, together with developing and debugging the dialog manager accounted for close to 50% of the total development time. The other half was accounted for (in fairly equal portions) by the development of the domain-specific system grammar and language generation templates, the development of the backend connections, and miscellaneous setup tasks. This evaluation is informal, does not capture the effort required for developing speech recognition and synthesis resources (the system was text-in / text-out) and addresses only the initial development stages. Nevertheless, we believe this exercise highlights both the flexibility and the rapid-development character of the RavenClaw dialog management framework.

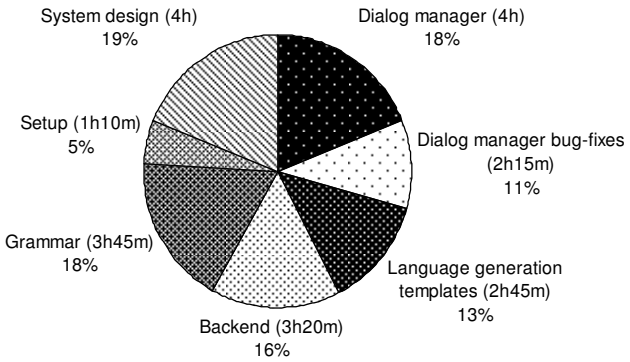


Figure 32. Wall-clock development time for various components of the Madeleine RavenClaw/Olympus system

3.4.8 ConQuest

ConQuest (**Con**ference **Quest**ions) [11] is a spoken dialog system that provides technical schedule information during scientific conferences, and the latest addition to the RavenClaw/Olympus family of systems. ConQuest facilitates access to information about papers and their authors, sessions, topics, special events, as well as various announcements made throughout the conference. Users can query for particular papers by specifying an author, a topic, session name, venue, or date and time. Similarly, users can query for sessions by topic, date and time, or venue. Once a session of interest is identified, the system can provide more details about the various papers presented in that session. In addition, the system can provide information about the social events associated with the conference, and allows participants to cast their votes for a People's Choice Best Paper Award. A sample interaction with the system is available in Appendix A.

The system implements an information access interface to the conference technical schedule. The interaction is largely slot-filling, with a few twists: the system has several back-off and constraint relaxation strategies. The user can take the initiative at any point in the dialog and switch between various topics of interests (e.g. from talking about papers to finding information about social events, to casting a vote for the best paper award, etc.) No significant challenges were encountered in implementing this interaction type in the RavenClaw dialog management framework.

The first version of this system, Conquest-IS06, was deployed during the InterSpeech-2006 conference in Pittsburgh, PA. Like the majority of RavenClaw/Olympus systems, Conquest-IS06 was configured with two parallel gender-specific Sphinx-II recognition engines. The vocabulary size was 4795 words, including 1492 author names, 659 lexicalized paper titles, 78 session names, 213 keywords and 7 room names. The semantic grammar used for language-understanding purposes contained 40 top-level grammar slots. The dialog task specification contained 73 dialog agents and 41 concepts. The dialog manager was configured with both explicit and implicit confirmations, and a wide array of non-understanding recovery strategies. The system used template-based language generation, and a Cepstral-based open-domain unit-selection synthesizer. Two versions of the system were available throughout the conference: a telephone version and a desktop-based system (available by the registration desk). A corpus totaling 174 sessions was collected during the conference. A new deployment is planned for the IJCAI-2007 conference in Hyderabad, India.

ConQuest not merely a dialog system that operates in a closed domain. Rather, ConQuest is better viewed as a "system template" that can be repeatedly instantiated for various conferences. The core resource needed to create a ConQuest instance is a database containing the technical schedule information. Based on this database, the corresponding vocabulary, grammar, language generation and synthesis resources can be semi-automatically generated, with relatively little manual effort. The dialog task specification is largely independent[9] of the conference database. While the development of the first system – ConQuest-IS06 – was similar to that of any other closed domain system, subsequent instantiations should be much easier to build. Once the ConQuest-IJCAI system is complete and deployed, it will be interesting to perform a comparative analysis of development time/effort and of the system's portability. We have released the ConQuest system (as well as the data collected with it so far) to the research community. We believe that in the future this system can provide a good platform and user base for research and comparative evaluations in spoken language interfaces.

3.4.9 Concluding remarks

In this section (3.4) we have described a number of conversational spoken language interfaces developed using the RavenClaw dialog management framework and the larger Olympus dialog system infrastructure. The systems operate across a variety of domains, including information-access, guidance

[9] Changes to the dialog task specification will be necessary only if new event types (beyond papers, sessions, keynote speech, social event) are present in the schedule database.

through procedural tasks, command-and-control, and message delivery. Three of these systems – RoomLine, Let's Go! Public, and ConQuest, have been successfully deployed into daily use.

The plan-based, hierarchical representation of the domain-specific control logic has easily accommodated each of these domains, indicating a high degree of versatility and scalability. Furthermore, the decoupling between the domain-specific dialog control logic and the RavenClaw dialog engine allows for dynamically extending and controlling the interaction plan at runtime. This property plays a key role in domains like LARRI, the Intelligent Procedure Assistant, and Madeleine, where the structure of the interaction is not known apriori, but rather generated on the fly, based on various back-end objects (i.e. procedural task in LARRI and the Intelligent Procedure Assistant, and medical diagnosis tree in Madeleine). The modularity of the RavenClaw implementation has also allowed us to easily port it into a different, open-agent-architecture based dialog system infrastructure in the Intelligent Procedure Assisstant system. Although further research is required (see more details in the following section), the framework was adapted to support a simple multi-participant conversation with a team of robots in the TeamTalk domain.

The high degree of flexibility afforded by the hierarchical plan-based dialog task specification is beneficial, especially if one is interested in building complex interactions. At the same time, we have learned through developing these systems that the increased expressive power of this representation can also pose a number of challenges. Consider for instance the less flexible finite-state paradigm for designing and specifying an interaction plan. In this case, the interaction plan is specified in a "positive" manner, by specifying possible transitions in the interaction: for instance, from state A the system can move to states B, C or D. Developing complex, mixed-initiative interactions in this formalism is in general a labor-intensive task: the number of possible transitions increases quickly with the total number of states. In contrast, in RavenClaw, the interaction plan is by default[10] defined in a "negative" fashion, by specifying constraints that need to always hold. At runtime, a path through the dialog task is generated, depending on the user inputs; anything can happen, as long as the constraints are satisfied. This allows for a mixed-initiative interaction and lessens development effort for complex interactions. At the same time, it means that multiple solutions are possible for any desired interaction. In addition, this can increase the amount of effort required to insure correct behavior under all possible inputs; debugging the system or tracing the reasoning behind dialog control decisions and outcomes can become a more difficult task. An interesting direction for future research would be to explore more closely the spectrum of possibilities between the two polarities we have mentioned above: the constraint-based task representation and the explicit state transition representation.

3.5 Summary and future work

In this chapter we described RavenClaw, a plan-based, task-independent dialog management framework. One of the key characteristics of the RavenClaw framework is that it enforces a clear separation between the domain-specific aspects of the dialog control logic and domain-independent conversational strategies. The domain-specific aspects are captured via a hierarchical plan for the conversation, provided by the system author. At the same time, a domain-independent dialog engine plans the conversation according to the specified logic and the user inputs, and automatically provides a rich repertoire of conversational skills, such as error handling, timing and turn-taking, context establishment, etc. This decoupled approach has a number of benefits: it lessens system development effort, it promotes reusability and portability of proposed solution, and it ensures consistency and uniformity in behavior both within and across systems. The hierarchical, plan-based representation of the domain-specific dialog control logic provides a high degree of versatility. To date, over a dozen

[10] Note that the framework allows the system developer to implement a different planning mechanism, including finite-state control in any of the agencies in the dialog task tree; this can be accomplished by overwriting the Execute routine for the desired agencies.

spoken dialog systems, operating in different domains, have been developed using the RavenClaw framework. Some of these systems have been deployed in day-to-day use. Overall, the framework has easily accommodated each of these domains and interaction styles.

Together with these systems, RavenClaw provides a robust platform for research in dialog management and conversational spoken language interfaces. Apart from the error handling work described in this dissertation, RavenClaw currently supports wo other research efforts. The first project, led by Harris and Rudnicky [47] aims to develop capabilities for multi-participant conversation. Most current spoken language interfaces are built for one-to-one conversation. However, new issues arise once we move to a multi-participant setting, where multiple agents can simultaneously engage in conversation. For instance, a number of problems regarding the conversation floor and the threading of subdialogs must be solved: how does each system identify who has the conversation floor and who is the addressee for any spoken utterance? How can multiple agents solve the channel contention problem, i.e. multiple agents speaking over each other? The second project, led by Raux and Eskenazi [86] investigates low-level interactional phenomena such as timing and turn-taking in conversation. Like the majority of other dialog management frameworks, the current version of RavenClaw makes a rigid "one-speaker-at-a-time" assumption. Although barge-in capabilities are supported, the dialog engine works asynchronously from the real-world: it does not use low-level information about the realization of various utterances; rather, utterances and actions are assumed to be executed instantaneously, as soon as they are planned. User barge-ins and backchannels are often interpreted in the incorrect context, and can lead to turn over-taking problems, and sometimes to complete interaction breakdowns. In order to investigate these issues, and enable better real-time reactive behaviors and more robust timing and turn-taking, Raux is currently developing a second version of the RavenClaw dialog manager and surrounding Olympus components that will take into account the precise timing of events perceived from the real-world and system actions. This novel architecture is currently deployed in the Let's Go! Public system and will enable research on low-level interactional phenomena.

The RavenClaw dialog management framework, together with a number of the systems described in section 3.4, provides the infrastructure for the error handling research conducted in this dissertation. Central for this work is the task-decoupled error handling architecture implemented as part of the RavenClaw dialog engine. In the next chapter, we describe this architecture in more detail.

Chapter 4

The RavenClaw error handling architecture

In this chapter we describe the key components of the error handling architecture underlying the RavenClaw dialog management framework. The architecture is task-independent and decouples both (1) the error handling strategies and (2) the error handling decision process from the actual dialog task specification. In doing so, it promotes reusability and learning, lessens the system authoring effort and ensures uniformity and consistency in behavior both within and across systems. Together with the larger, encompassing RavenClaw dialog management framework, this error handling architecture provides the infrastructure and experimental platform for the rest of the work described in this dissertation.

4.1 Background and objectives

The ability to accurately detect errors and recover from them is paramount in any spoken language interface. Although certain error detection mechanisms can operate early in the input processing stage, some errors such as no-match non-understandings can only be detected at the dialog management level. Only at this level, after the system attempts to integrate the decoded semantics of the current user turn into the larger discourse context, enough information is accumulated to make a fully informed judgment in an error detection (or diagnosis) task. Similarly, the decisions to engage in various error handling strategies such as confirming a concept, asking the user to repeat, asking the user to rephrase, etc. are made at the dialog management level. The system has to balance the costs of engaging in an error recovery strategy against those of continuing the interaction with potentially incorrect information. Error handling is therefore an important function in any dialog manager.

The responsibility for handling potential errors is often delegated to the system author. Error handling is regarded as an integral part of the system's dialog task. In this case, the system author has to write code that handles potential misunderstanding and non-understanding errors. This approach leads to monolithic one-time solutions that tend to lack portability, are prone to bugs and hard to maintain. Furthermore, they often result in inconsistent behaviors throughout different parts of the dialog.

The alternative is a decoupled, task-independent or "toolkit" [71] approach to error handling. We argue that this type of approach is feasible in the context of a plan-based dialog manage-

ment framework, and it creates a number of advantages. We have already discussed in the previous chapter the benefits that result from insulating the domain-independent aspects of the dialog control logic from the domain-independent ones. We believe that, for a wide class of task-oriented systems, error handling can and should be regarded as a domain-independent conversational skill, and can therefore be decoupled from the domain-specific aspects of dialog control logic, i.e. from the dialog task specification. In general, a large number of error recovery strategies, such as asking the user to repeat or rephrase, are entirely task-independent. Other more complex strategies, for instance explicit and implicit confirmations, can also be decoupled from the task by using appropriate parameterizations, for instance explicitly confirm concept `foo`.

A task-independent approach has several benefits. First, it increases the degree of consistency in the interaction style, both within and across tasks. This in turn leads to a better user experience, and facilitates the transference of learning effects across systems. Second, the approach significantly decreases development and maintenance efforts; it fosters reusability and ease-of-use. Ideally, system authors should not have to worry about handling understanding-errors[11]. Rather, they should simply focus on describing the domain-specific dialog control logic, under the assumption that inputs to the system will always be perfect. At the same time, the dialog engine should automatically prevent understanding-errors or gracefully recover from them. Last but not least, a task-independent approach represents a more sensible choice from a software engineering perspective.

The case for separating out various task-independent aspects of the conversation has in fact been made previously. Balentine and Morgan [5] recommend identifying a set of generic speech behaviors and constructing dialogue systems starting from these basic building blocks. Lemon et al [65] propose a dialog management architecture where "content-level communicative processes" and "interaction-level phenomena" are handled in separate layers. The Universal Speech Interface project [99, 122, 123] proposes to identify and leverage a basic set of dialog universals that transfer across applications and domains. Error handling capabilities are also a good candidate for this type of decoupling, and can be implemented as domain- and task-independent conversational mechanisms.

In this chapter, we describe the practical implementation for a decoupled, task-independent error handling architecture in the context of a complex, hierarchical plan-based dialog management framework (RavenClaw). This error handling architecture provides the infrastructure and experimental platform for the research described in the rest of this dissertation. Apart from task-independence, several other objectives were pursued in its development:

Modularity. The error handling architecture should be modular, i.e. it should encapsulate and separate the mechanisms for detecting errors, the error recovery strategies, and the error recovery policies. The architecture should allow independent access to each of these components: it should be easy to develop and plug in new mechanisms for detecting or diagnosing errors, new error recovery strategies, or new error recovery policies.

Ease-of-use. The error handling architecture should be easy-to-use and should lessen the system authoring effort. Ideally the dialog engine should ensure automatically that there are no understanding-errors and that the dialog advances normally towards its goals. System authors should mostly focus on describing the domain-specific aspects of the dialog control logic. Additionally, they should be able to customize at a high-level the error handling behaviors. For instance system authors should be able to specify the set of strategies to be used for recovery, and other behavioral parameters like for instance how conservative the system should be in error handling.

Reusability. The proposed error handling architecture should promote the reusability of various error handling components (e.g. detection mechanisms, recovery strategies, recovery policies) across domains.

Adaptability. The performance of the underlying speech recognition and language under-

[11] Here, we refer to understanding-errors rather than domain-specific errors, e.g. booking a flight between the same arrival and destination cities in a flight reservation system; the burden for handling this latter type of domain-specific errors remains with the system author.

standing components can vary dramatically across different domains and systems. As a consequence, we are interested in developing data-driven, learning-based error handling solutions that are adapted to the particular characteristics of the domain in which the system operates. The architecture should allow for automatic tuning of error handling behaviors and should facilitate learning from experience, preferably without requiring large amounts of developer supervision.

Scalability. The proposed error handling architecture should scale well to real-world, practical spoken language interfaces.

We begin by presenting a high-level overview of the proposed error handling architecture in the next section. Then, in sections 4.3 and 4.4, we discuss the mechanisms for detecting and recovering from misunderstandings and non-understandings. Finally, in section 4.5 we present concluding remarks.

4.2 Architecture

4.2.1 Organizing principles

The error handling architecture in the RavenClaw dialog management framework subsumes two main components: (1) a set of error recovery strategies, and (2) an error handling decision process that engages these strategies at the appropriate time (see Figure 33.)

The error recovery strategies fall into two categories: (1) strategies for recovering from misunderstandings such as explicit and implicit confirmation, and (2) strategies for recovering from non-understandings such as asking the user to repeat, asking the user to rephrase, providing help, etc. These strategies were authored using the RavenClaw dialog task specification formalism described in the previous chapter; they are available as library dialog agencies. System authors simply specify which strategies should be used by the dialog manager, and configure them accordingly (later on, we will describe the various parameters these strategies take.) Furthermore, system authors may also develop additional error handling strategies and plug them into any new or existing RavenClaw-based system.

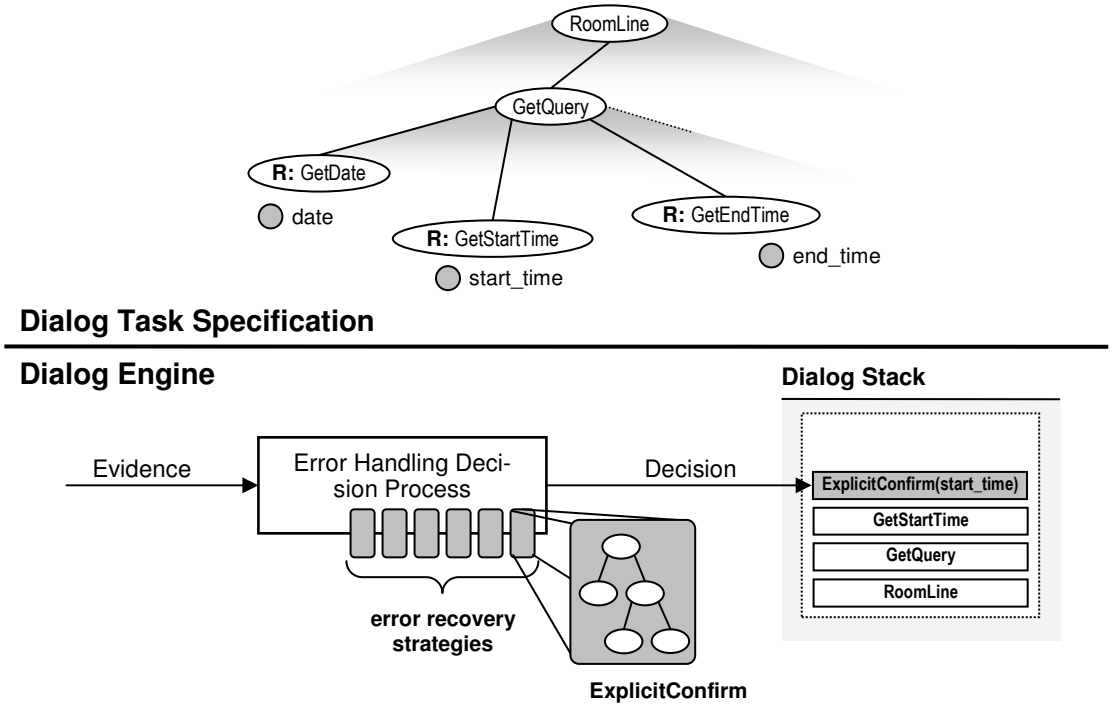


Figure 33. RavenClaw error handling architecture - block diagram

The responsibility for handling potential errors is delegated to the error handling decision process, a subcomponent of the RavenClaw dialog engine. At each point in the dialog (more precisely during each Execution Phase), the error handling decision process collects available evidence and decides which error recovery strategy (if any) should be engaged. If action is deemed necessary, the error handling decision process creates an instance of the corresponding error recovery strategy, parameterizes it accordingly, and pushes it on the dialog stack. In effect the error handling decision process changes the dialog task to ensure that potential errors are handled accordingly. For instance, in the example illustrated in Figure 33, the system decided to engage an explicit confirmation for the `start_time` concept (more details about how these decisions are made will be presented later in this chapter.) The system therefore instantiated an `ExplicitConfirm` agency (which implements an explicit confirmation strategy), parameterized it by passing a pointer to the concept to be confirmed (in this case `start_time`), and placed it on the dialog stack. Next, the strategy executes. Once completed, it is removed from the stack and the dialog resumes from where it left off. During the execution of the explicit confirmation, all other dialog control mechanisms are still in place; for instance, the user could request more help, or even shift the current dialog topic.

This design, in which both the error recovery strategies and the error handling decision process are decoupled from each other as well as from the actual dialog task specification has a number of benefits. First, it significantly lessens the system development effort. System authors are free to write the dialog task specification (i.e. the domain-specific aspects of the dialog-control logic) under the assumption that the inputs to the system will be understood correctly. The responsibility for ensuring that the system maintains accurate information and that the dialog advances normally towards its goals is delegated to the error handling decision process in the RavenClaw dialog engine. The error handling process will modify the dialog task dynamically, engaging various strategies to prevent and recover from errors. Second, the proposed architecture promotes the reusability of error handling strategies across different systems. A large set of error recovery strategies are currently available in the RavenClaw dialog management framework. These strategies, together with any new strategies developed by a system author can be easily plugged into any new or existing RavenClaw-based spoken dialog system. Lastly, the approach ensures uniformity and consistency in the system's behavior, both within and across systems.

4.2.2 Distributed error handling decision process

The error handling decision process is implemented in a distributed fashion. It consists of a collection of smaller **error handling models**, automatically associated with each request agent and each concept in the dialog task tree, as illustrated in Figure 34.

The error handling models associated with individual concepts, also known as **concept error handling models**, are in charge of recovering from misunderstandings on those concepts. They use as evidence confidence scores for that particular concept (i.e. the current belief over that concept) and engage the misunderstanding recovery strategies such as explicit or implicit confirmation.

The error handling models associated with individual request-agents, also known as **request error handling models**, are in charge of recovering from non-understandings that occur during the corresponding requests. They use as evidence features characterizing the current non-understanding and dialog state and engage the non-understanding recovery strategies, such as asking the user to repeat, asking the user to rephrase, repeating the system prompt, providing help, etc.

During the error handling phase, each concept- and request error handling model computes and forwards its decision to a gating mechanism. The gating mechanism queues up these actions (if necessary) and executes them one at a time. For instance, in the example from Figure 34, the error handling model for the `start_time` concept signaled that it wanted to engage an explicit confirmation on that concept; the other local models did not take any action. In this case the gating mechanism created a new instance of an explicit confirmation agency, passed it a pointer to the concept to be confirmed (`start_time`), and placed it on the dialog stack, as illustrated in Figure 33. On completion, the belief over the confirmed concept is updated in light of the user response, and the dialog

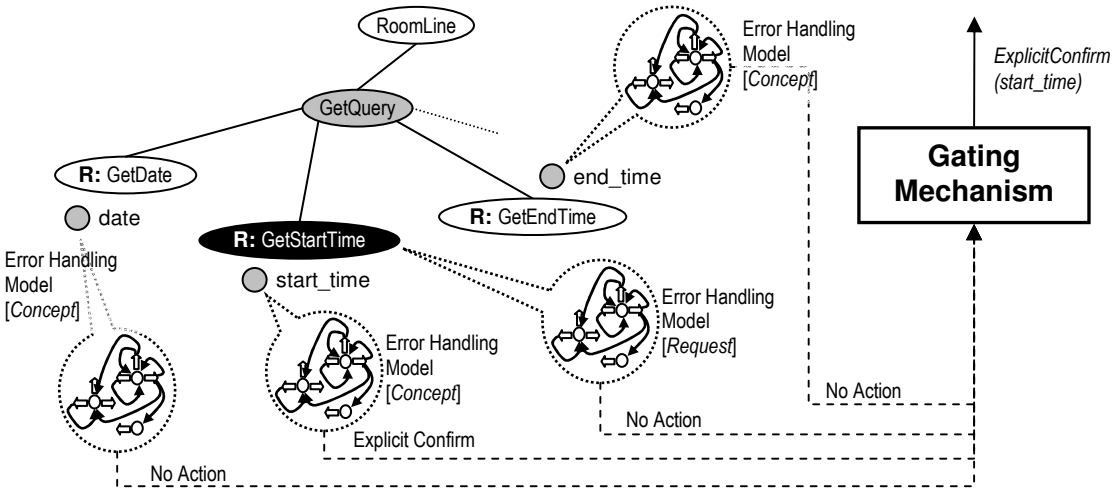


Figure 34. Distributed error handling decision process

resumes from where it left off.

Several implementations for the local error handling models are available in the RavenClaw dialog management framework. They are discussed in more detail in the sections 4.3 (for concept error handling models) and 4.4 (for request error handling models). The behavior for each error handling model is specified by means of a **control-policy**. Typically, a **control-policy** defines the model's behavior via a set of parameters that describe the cost of various actions under various circumstances. (Other parameters might also be used, depending on the model structure; more details will be presented in the following sections.) For instance a predefined "pessimistic" policy for a concept-level error handling model specifies a high cost for false-acceptances; as a result the model tends to always engage in explicit confirmations. Alternatively an predefined "optimistic" policy specifies a lower relative cost for false-acceptances, and as a result the model engages in explicit confirmations only if the confidence for the top concept hypothesis is below a certain threshold. A number of predefined error handling models and associated control policies are available in the RavenClaw dialog management framework. When developing a new system, system authors must specify the type of error handling model and corresponding policy for each concept and request agent in the dialog task tree (defaults are also available). In addition, system authors can also define new control policies, or implement new error handling models altogether.

The distributed and encapsulated nature of the error handling decision increases its scalability and favors learning-based approaches to error handling. First, the structure and parameters of individual error handling models can be tied across different concepts or request-agents. In the example from Figure 34, the error handling model for the `start_time` concept can be assumed to be identical to the one for the `end_time` concept; all models controlling Boolean (Yes/No) concepts could be also tied together. Parameter tying can greatly improve the scalability of learning-based approaches because the data is polled together and the total number of parameters grows sub-linearly with the size of the task (i.e. with the number of concepts and request-agents in the dialog task tree). Secondly, policies learned in a system can be reused in other systems because the error handling models are decoupled from the actual dialog task specification. For instance, we expect that the grounding of yes/no concepts functions similarly at different locations in the dialog, but also across domains. Thirdly, the proposed architecture accommodates dynamic task generation. The dialog task tree (the dialog plan) can be dynamically expanded at runtime, and the corresponding concept- and request- error handling models will be created on the fly. If the model structure and parameters are tied, we do not need to know the full structure of the task to be executed by the system in advance.

These advantages stem from an independence assumption made by the proposed architecture: the error handling models associated with different concepts and request-agents in the dialog task tree operate independently of each other. In some dialog tasks, long-range dependencies might

exist; in these cases, the independence assumption is violated. Such long-range dependencies can be addressed to a large extent by including global information (e.g. dialog history information, inter-concept dependencies etc.) in the local error handling models. In fact, in Chapter 6 and Chapter 8 we will discuss a number of error handling models that do incorporate such global information. The view taken in this work is that error handling can (to a large extent) be modeled as a local process. I believe that overall the advantages gained by making this independence assumption and resorting to a distributed and encapsulated error handling process significantly outweigh the potential drawbacks.

In the remaining two sections of this chapter, we discuss the structural and functional details of the concept- and request error handling models. We begin with the concept error handling models.

4.3 Concept error handling models

In the RavenClaw error handling architecture, a concept error handling model is automatically associated with every concept in the dialog task tree. Each model makes error handling decisions with respect to the corresponding concept. The models insure that the information contained in concepts is accurate; they can engage various strategies to recover from potential misunderstandings. The concept error handling models represent the system's line of defense against misunderstandings.

In this section, we describe the structure and function of concept error handling models in more detail. We begin by discussing the concept-level belief representation and RavenClaw belief-updating mechanisms, since they play a key role in these models. Next, in subsection 4.3.2, we discuss the strategies used by these models to recover from misunderstandings. Lastly, in subsection 4.3.3, we discuss the models' internal structure and the decision-making mechanisms used by these models, and we describe the set of predefined control-policies.

4.3.1 Belief representation and belief updating

The concept-level error handling model relies heavily on the advanced concept-level belief representation discussed in subsection 3.2.2.3 from the previous chapter. RavenClaw represents belief in a concept via a set of possible values for that concept and their corresponding confidence scores. For instance, the belief over the `departure_city` concept in a flight reservation system might be described at some point in the dialog as `departure_city = {Boston/0.43; Aspen/0.17}`. Initially, these concept-level beliefs are constructed based on the incoming recognition results and corresponding confidence scores (potentially using information from an n-best list). Throughout the dialog, the beliefs are updated based on information from subsequent user responses to various system actions. For instance, if the system attempts to explicitly confirm a top concept hypothesis (e.g. `Boston` in the example above) and hears another concept hypothesis in the user response (e.g. `Austin`), a new updated belief for the `departure_city` concept will be assembled – see Figure 35.

The concept error handling model bases its decisions to engage in various confirmation strategies on the concept belief. As a consequence, the belief representation and belief updating calculus play an essential role in concept-level error handling. The more accurate the constructed beliefs are, the better the model's decisions will be.

Most spoken language interfaces do not track multiple alternate values at the concept level. Typically, systems store a single current value for a concept, together with a confidence score, or a confidence indicator (e.g. low, medium, high). Concept updates are generally performed based on simple heuristic rules. For instance, a commonly encountered rule is to let new values overwrite old values. The intuition behind this rule is that users will correct the system throughout the conversation, and therefore trusting the last value heard is better than keeping an old value. Additionally, specialized rules are defined for the confirmation cases: if the system hears a yes-type answer (e.g. yes, that's correct, that's right, etc.) to an explicit confirmation action, it boosts the confidence for the confirmed value to very high; if the system hears a no-type answer, it deletes the current concept hypothesis altogether.

The RavenClaw dialog management framework can store and track multiple concept-level hypotheses. It supports the traditional, heuristic-rule based approach to belief updating. In addition, RavenClaw also supports a novel, data-driven belief updating mechanism that significantly outperforms the heuristic approach. Below, I present the heuristic belief updating mechanism in the RavenClaw dialog management framework, and briefly outline the more advanced, data-driven approach. (This second approach constitutes one of the main contributions of this dissertation and is discussed in depth in Chapter 6.)

4.3.1.1 Heuristic belief updating

The heuristic belief updating mechanism in the RavenClaw dialog management framework relies on a natural, naïve probabilistic update rule.

When the concept is empty and a first potential value is obtained from a recognition result, the system constructs an initial belief by considering that value in conjunction with the incoming recognition (or semantic) confidence score. For instance, in the example from Figure 35, the system constructs the initial belief `departure_city = {Boston/0.35}` by taking into account the first decoded response from turn 2 and the corresponding utterance-level confidence score. The remaining probability mass is considered to be distributed uniformly across all other potential hypotheses for that concept. For instance, if there are 500 possible departure cities, the probability for each of the 499 remaining cities is assumed to be 0.65/499=0.0013. RavenClaw does not actively store this full probability distribution; instead, it stores `departure_city = {Boston/0.35}`, the cardinality of that concept, and works with the assumption that the rest of the mass is distributed equally among all other hypotheses.

To update its beliefs throughout the conversation, the system "multiplies" the probability distribution for its current (initial) belief with the probability distribution corresponding to the new recognition hypothesis, and renormalizes. For instance, in Figure 35 the system engaged in an explicit confirmation in turn 3 and obtained a user response (Austin). The initial belief it has over the `departure_city` concept contains `Boston` with confidence 0.35 and all other cities with confidence 0.0013. The distribution corresponding to the recognition result contains `Austin` with confidence 0.57 and all other cities with confidence 0.00086. These two distributions are multiplied and the resulting distribution is renormalized. In this case the updated belief will be `departure_city = {Austin/0.4631; Boston/0.1881}`.

Throughout these computations, the RavenClaw heuristic belief updating mechanism ensures that the probability mass for all hypotheses heard so far does not exceed 0.95. This extra constraint is imposed to ensure that the unheard hypotheses still receive some probability mass. In the absence of this constraint, if the system repeatedly heard the same concept value in successive user turns, the probability mass for the unheard hypotheses would drop to a very low value. This would prevent those values from gaining enough mass to compete with the previously heard hypothesis, even if they started appearing again in the recognition results. The need for this extra constraint already hints at the limitations and ad-hoc nature of heuristic approaches to belief updating. In Chapter 6 we will describe a data-driven solution to this belief updating problem that addresses this as well as other shortcomings.

In addition, two other heuristic rules are used to update the system's beliefs after explicit and implicit confirmation actions. If the user response to an explicit confirmation contains a positive marker (e.g. *yes*, *that's right*, *that's correct*, etc.), the confidence score for that concept value is set to the maximum possible value, 0.95, and the alternate hypotheses (if any) are deleted from the belief space; the framework still considers the remaining probability mass, 0.05, to be equally distributed among all other possible values. If the user response to an explicit confirmation contains a negative marker (e.g. *no*, *that's wrong*, etc.), the corresponding hypothesis is deleted from the belief space; its probability mass is therefore assumed to be uniformly reassigned across all the unheard hypotheses.

After implicit confirmations, the system updates the confidence score for the confirmed hypothesis to 0.95 if no negative marker or new concept value is found in the user response. The sys-

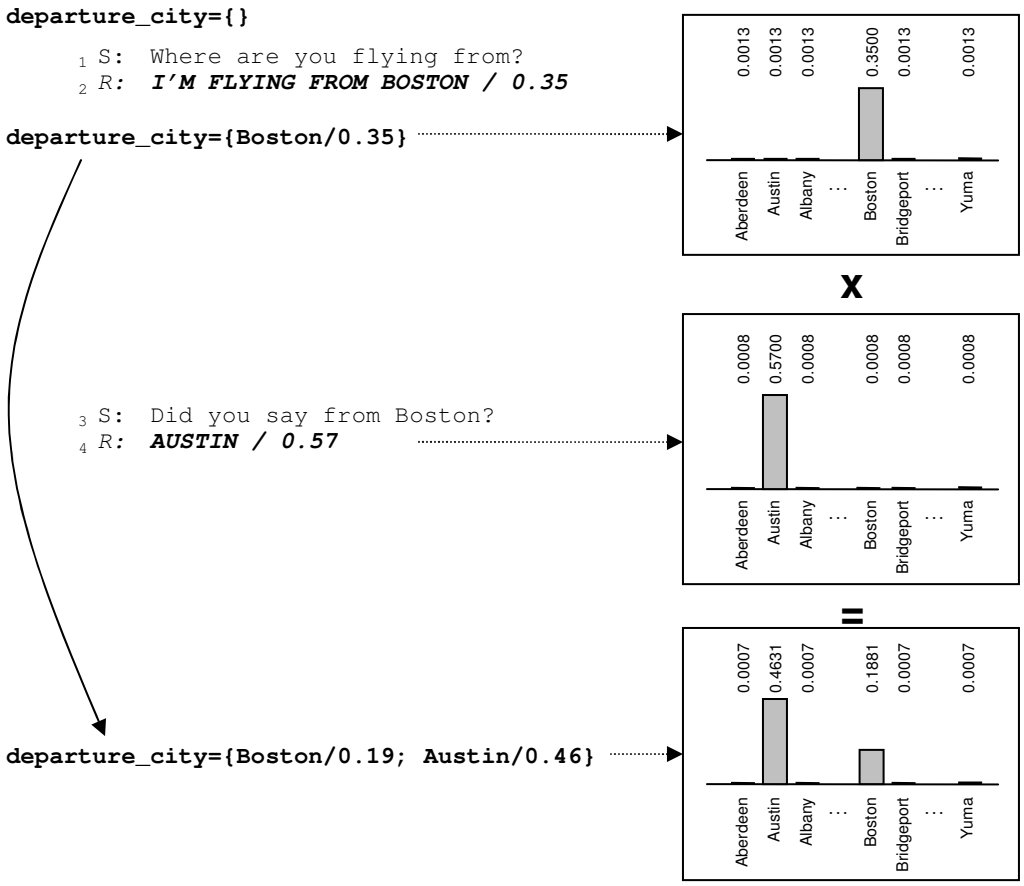


Figure 35. Heuristic belief updating in the RavenClaw dialog management framework

tem assumes that if user is not contradicting or correcting the system, the value is correct. If a negative marker is found in the user response (and the system action following the implicit confirmation is not a yes/no question), the system deletes the hypothesis from the belief space, like after an explicit confirmation. Finally, if another value for the concept appears in the decoded response, the system relies on the naïve probabilistic update rule described above.

4.3.1.2 Data-driven belief updating models

The heuristic approach to belief updating described in the previous section has a number of shortcomings. First, looking for simple confirmation and disconfirmation markers after explicit and implicit confirmation actions is insufficient. User responses to these system actions can go beyond simple yes/no answers, as we shall see in the empirical analysis from Chapter 6, subsection 6.4.3. Furthermore, user responses are also subject to speech recognition errors. Naively multiplying the probability distributions can artificially inflate the confidence for a repeatedly misrecognized item.

In Chapter 6, we describe an alternative, data-driven method for performing belief updates. The proposed method is one of the main contributions of this dissertation, and is allocated an entire chapter by itself. In short, the belief updating problem is cast as a supervised learning task: given an initial belief, a system action and a set of features that characterize the user response, construct an updated belief that is as accurate as possible. Experimental results indicate that the proposed data-driven approach significantly outperforms the heuristic rules described previously, and leads to large improvements in the overall effectiveness and efficiency of the interaction.

4.3.2 Recovery strategies

The belief updating mechanisms described above provide the basis for decision making in the concept error handling models; the decision to engage in one recovery strategy or another is based on the confidence scores for the different concept hypotheses. Next, I will discuss the strategies that can be engaged by these models to recover from potential misunderstandings. Currently, two such strategies are available in the RavenClaw dialog management framework: **explicit confirmation** and **implicit confirmation**.

In an **explicit confirmation**, the system asks the user a yes/no question, in an effort to directly (or explicitly) confirm a certain concept value. For instance: "Did you say from Boston?" – see Figure 35. Alternatively, in an **implicit confirmation**, the system echoes back the value it heard to the user, and continues with the next question; the assumption is that, if the value is incorrect, the user will detect the error and interject a correction – see Figure 36. Below, I discuss these two strategies in more detail. Although not currently available, other strategies for recovering from misunderstandings, such as disambiguation[12] could also be easily implemented and used in the RavenClaw dialog management framework.

4.3.2.1 Explicit confirmation

Like all other error recovery strategies in the RavenClaw dialog management framework, the explicit confirmation strategy is decoupled from the dialog task specification. The strategy is implemented as a library request-agent. At runtime, when the concept error handling model decides to engage in an explicit confirmation, an instance of this explicit confirmation request-agent is created. A pointer to the concept to be confirmed is passed to the explicit confirmation request-agent, which is then placed on top of the execution stack. In effect, the dialog engine automatically creates a sub-dialog for confirming the concept value. When executed, the explicit confirmation agent requests a Boolean `confirm` concept, and expects a yes- or no-type answer (using the `[Yes]` and `[No]` generic grammar slots). For generating the request, the agent issues an `explicit_confirm` prompt for the corresponding concept; this prompt must be supplied by the system author at the language generation level. Throughout the explicit confirmation sub-dialog, all the other dialog mechanisms remain in effect. For instance, the user could still over-answer the system's explicit confirmation question (e.g. "yes from Boston to San Francisco"), ask for help, attempt to shift the conversation topic, etc.

Once a response is received and the belief over the concept is updated, the explicit confirmation agent completes and is eliminated from the dialog stack. The dialog resumes from where it left off. Note that, depending on the updated belief and the error handling policy, the system might subsequently engage in a new confirmation action on the same concept. For instance, in the example from Figure 35, the system might continue with another explicit confirmation on the same concept, e.g. "Did you say Austin?"

Three other parameters can be used to control the behavior of the explicit confirmation strategy. First, the `confirmation_lm` parameter can specify a language model to be used when the explicit confirmation agent is in focus (since in general yes/no answers are expected to explicit confirmation questions, a language model focused on this type of answers might be more appropriate). Secondly, the strategy can be configured to accept DTMF (touch-tone) inputs (e.g. press 1 for 'yes', or 0 for 'no'). The last parameter, `grounding_model`, specifies the error handling model and control-policy for the request agent that implements the explicit confirmation. This model governs recovery from non-understanding errors that might occur during the system's explicit confirmation action (request-level error handling models are discussed later in this chapter, in section 4.4.)

4.3.2.2 Implicit confirmation

[12] in a disambiguation the system tries to determine which one of two concept hypotheses is the correct one. For instance: "Did you say Boston or Austin?"

The implicit confirmation strategy is implemented by means of a pair of library dialog agents. The first agent is an inform agent that echoes back the value-to-be-confirmed to the user. It does so by issuing an `implicit_confirm` prompt that must be provided by the system author at the language generation level. The second agent is an expect-agent that captures a potential confirmation or disconfirmation from the user. At runtime, when the concept error handling model decides to trigger an implicit confirmation, an instance for each of these agents is created. A pointer to the concept undergoing the confirmation is passed to each agent. The agents are placed on top of the execution stack. The inform agent executes, issues the implicit confirmation prompt, and then completes and is eliminated from the dialog stack. Since the expect agent is not executable, the dialog planning continues from where it left off. The expect agent remains however on the execution stack, and will define its expectations on the agenda during the next Input Phase. Based on the follow-up user response, RavenClaw updates the belief over the confirmed concept. The expect agent also completes once the follow-up user response is received.

```
1 S:  flying from Boston... where would you like to go?
2 U:  no from Austin
  R:  NO FROM AUSTIN / 0.75
```

Figure 36. A sample implicit confirmation for the departure city concept

4.3.3 Structure and control-policies

So far, we have discussed the belief updating mechanisms and the confirmation strategies available to the concept error handling model. Next we turn our attention to the structure of the concept-level error handling model, which ties the two together: based on the constructed beliefs, it decides which strategy should be engaged at any given point in the dialog.

The decision of which strategy to engage is predicated on the likelihood of a misunderstanding (captured by the confidence scores in the concept belief representation) and on the trade-offs between the explicit and implicit confirmation strategies. These trade-offs are relatively well understood. During an explicit confirmation, the system takes an extra dialog turn to confirm a concept value; in general explicit confirmation questions lead to simple user responses, such as *"yes"*, *"no"*, or equivalents. At the same time, excessive use of explicit confirmations can significant increase dialog length and also user frustration. In contrast, implicit confirmations do not take an extra dialog turn; but responses to implicit confirmations can be more difficult to handle. When the system attempts to implicitly confirm an incorrect value, it increases the cognitive load on the user [62], who now has to choose between formulating a correction, providing an answer to the next question, or both. In general, user responses to implicit confirmations span a wider language spectrum and are more difficult to recognize and understand correctly (see [63] as well as our own empirical analysis from Chapter 6, subsection 6.4.3). This can also further increase the difficulty of the belief updating task. Given these trade-offs, explicit confirmations are typically used when the system is more unsure about the value to be confirmed. Implicit confirmations are used when the system is fairly confident that the value to be confirmed is correct. In the latter case, the cost for the implicit confirmation is lower than for an explicit confirmation: the user can respond to the system's follow-up prompt and the dialog can advance normally[13], without the loss of an extra turn.

Based on these trade-offs, a typical control-policy for handling potential misunderstandings is illustrated in Figure 37: if the confidence score for the top concept hypothesis is very high, above a certain threshold t_1, accept that value as correct, i.e. consider it grounded. If the confidence for the top concept hypothesis is medium-high, i.e. below t_1 but above another threshold t_2, then engage in

[13] a caveat here is that users might back-channel an acknowledgement immediately after the system's implicit confirmation, before they provide an answer to the follow-up system prompt; in this situations the system must be ready to absorb this backchannel

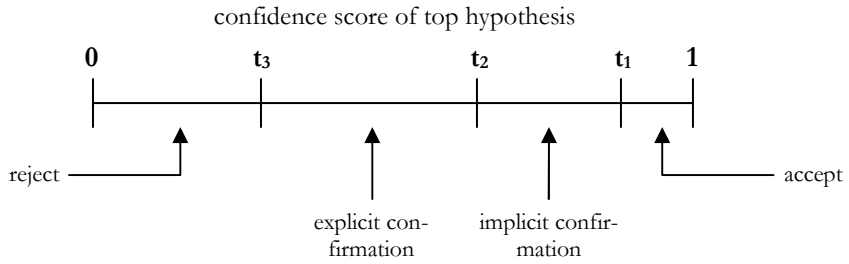


Figure 37. A typical control policy for engaging concept-level confirmation strategies

an implicit confirmation. Alternatively, if the confidence score is medium-low, i.e. below t_2, but above another threshold t_3, then engage in an explicit confirmation. Finally, if the confidence is very low, below t_3, then reject that concept value altogether.

The default concept error handling model available in the RavenClaw dialog management framework implements this type of control policy. Currently, RavenClaw deals with rejections separately, at the utterance, rather than at the concept level. The concept-level error handling model can therefore engage one of three actions: accept, explicit confirmation, and implicit confirmation. In the following subsection, we describe in more detail the implementation of the default concept error handling model, and some of the predefined control-policies.

4.3.3.1 The default concept-level error handling model

The default concept-level error handling model in RavenClaw is currently implemented as a Markov decision process (MDP). An MDP is defined by a discrete set of states $\{s_i\}_{i=1..n}$, a number of actions available from each state $\{a_j\}_{j=1..k}$ and a set of transition probabilities that stochastically define which state the model moves to once an action is engaged $P(s_{t+1} \mid s_t, a_p)$.

The structure of the default concept error handling model is illustrated in Figure 38. The model has 4 states: `inactive`, `correct`, `incorrect` and `grounded`. The model is in the `inactive` state when the corresponding concept is empty, i.e. the system does not yet have a candidate hypothesis for this concept. The model is in the `correct` state if the current top hypothesis for the concept is indeed correct. Alternatively, if the current top hypothesis is incorrect, the model is in the `incorrect` state. Finally, the model is in the `grounded` state when the current top hypothesis for the concept is considered grounded (this happens once the system takes an accept action and the concept is not empty). There are 3 actions that the model can engage at any given point in the dialog: `accept`, `explicit_confirm` and `implicit_confirm`. The accept action is only available from the `inactive`, `correct` and `incorrect` states; when engaged from the `inactive` state it does nothing; when engaged from the `correct` or `incorrect` state it marks the concept as grounded. The `explicit_confirm` and `implicit_confirm` actions are available only from the `correct` and `incorrect` states – see Figure 38; they engage the corresponding confirmation strategies.

At each point in the dialog, the state of the concept error handling model is inferred from the belief over the corresponding concept. Note that the system does not know whether the current top hypothesis is correct or incorrect. Rather, it knows a probability of that value being correct or not (this probability is extracted from the belief over that concept: it is given by the confidence score for the current top hypothesis.) The concept error handling model is therefore a partially-observable Markov decision process: at every point in the dialog, rather than knowing the precise state, the error handling model only knows a probability distribution over the 4 states (i.e. a belief state): $<p_{inactive}, p_{correct}, p_{incorrect}, p_{grounded}>$. Given the structure of the model, the set of belief states is restricted to a certain subspace of that simplex:

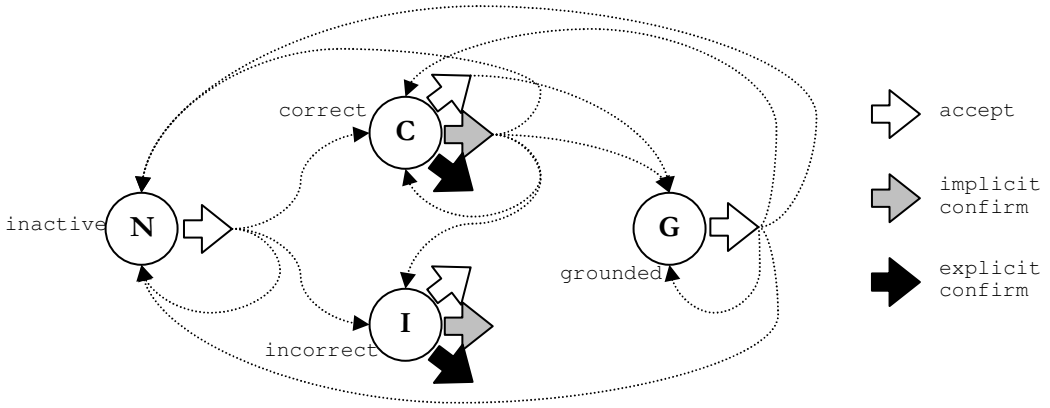


Figure 38. Structure for the default concept-level error handling model (not all transitions are illustrated)

- initially, when no value has yet been heard for the concept, the error handling model is in the `inactive` state: $<p_{inactive}=1, p_{correct}=0, p_{incorrect}=0, p_{grounded}=0>$; the model will therefore engage the only available action, `accept`, which in this case has no effect.
- if the belief over the concept contains a top hypothesis, the belief state for the model is a mixture of the `correct` and `incorrect` states $<p_{inactive}=0, p_{correct}, p_{incorrect}, p_{grounded}=0>$, where $p_{correct}$ is given by the confidence score of the current top hypothesis and $p_{incorrect} = 1-p_{correct}$. In this case, the system may decide to engage in an `accept`, `explicit_confirm` or `implicit_confirm` action. If the system engages in an `accept` action, the concept is marked as grounded (a binary flag is set on the concept indicating that the concept has been grounded.); otherwise, one of the corresponding confirmation strategies is engaged.
- finally, if the concept has been already marked as grounded, the corresponding error handling model is in the `grounded` state: $<p_{inactive}=0, p_{correct}=0, p_{incorrect}=0, p_{grounded}=1>$.

The transitions from one state to another are not modeled directly by the concept error handling model. The belief state for the concept error handling model is recomputed at each point in the dialog, based on the current belief for the corresponding concept. The belief updating process (which takes into account the user inputs) governs the actual dynamics of the model.

The behavior of the concept error handling model is specified by means of a control-policy that defines which action the model should engage from every state. The control-policy is specified by providing the utility for each action in each state. An example policy is presented in Figure 39. The only available action from the `inactive` and `grounded` states is `accept`. If the concept value is correct, the utility of accepting that value and therefore considering it grounded is 2; the utility of performing an explicit confirmation is -4 and the utility of performing an implicit confirmation is 0 (explicit confirmation is more costly since it takes an extra dialog turn). If the concept value is incorrect, accepting it has a high cost (-4), while explicit confirmation would be the most useful action (4). At runtime, the system computes the overall expected utility for each action by taking into account the state probabilities and the defined utilities. These utilities induce a control-policy that in fact corresponds to a classical threshold-on-confidence-score model. For instance, in the example shown in Figure 39, the model will engage an `accept` action if the confidence score is above 0.8; alternatively, if the score is between 0.44 and 0.8, the model will engage the implicit confirmation strategy and if the score is below 0.44 the system will engage the explicit confirmation strategy.

At this point, the reader might ask: why not just use a simpler model? As we have seen just a pair of thresholds could implement the same control policy; also, the policy would be easier to specify. Such a model could be easily implemented in the RavenClaw dialog management framework and used wherever appropriate. There are a number of reasons the MDP representation was chosen. First, this representation brings the costs for various actions to the forefront. In general, the thresh-

	accept	explicit confirm	implicit confirm
inactive	10	N/A	N/A
correct	2	-4	0
incorrect	-4	4	1
grounded	10	N/A	N/A

```
exploration_mode=epsilon-greedy
exploration_param=0.1
```

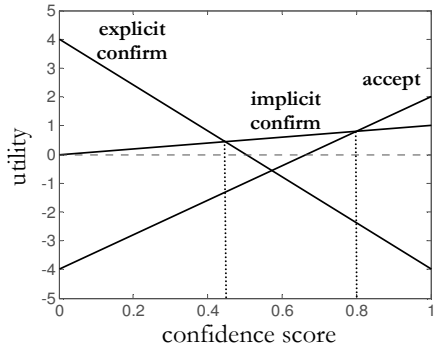


Figure 39. Control policy for concept-level error handling model (right-hand side); thresholds induced by control policy (left-hand side)

olds themselves are tuned based on relative costs for actions, and the current error handling model implementation makes this process transparent. Perhaps more importantly, the MDP implementation allows us to explore various learning-based approaches to the problem of deriving a control policy. In so far we have assumed that the model chooses the action with the highest expected utility. However, while collecting data for learning it is desirable not only to exploit the current maximum utility action, but also to explore how some of the other apparently suboptimal actions would perform under the current circumstances (the exploration-exploitation trade-off). The MDP-based implementation enables exploratory policies via the `exploration_mode` parameter. This parameter controls how the model chooses between different actions, once the expected utilities are computed. The following options are available:

- `greedy`: the model chooses the action with the highest expected utility.
- `epsilon-greedy`: with probability 1-epsilon, the model chooses the action with the highest expected utility; with probability epsilon the model chooses a random action; epsilon is specified via the `exploration_param` parameter.
- `stochastic`: the model chooses an action with a probability proportional to the expected utility for that action (in this case the utilities must be positive).
- `soft-max`: the model uses a soft-max (Boltzmann) heuristic for choosing between different actions. The probability for engaging an action is:

$$P(A_k) = \frac{e^{U(A_k)/T}}{\sum_k e^{U(A_k)/T}},$$

where $U(A_k)$ is the utility for action A_k, and T is a temperature parameter specified via the `exploration_param` parameter.

A number of predefined control-policies for concept error handling models are available in the RavenClaw dialog management framework. For instance, a predefined **pessimistic** policy sets the costs such that the accept and implicit confirmation thresholds are very high; in contrast an predefined **optimistic** policy set these thresholds at lower levels. Finally, an **explicit-only** policy sets the costs such that the system always explicitly confirms. System authors can define and use additional policies by providing the costs/utilities for the different actions in different states. An interesting question is: can the system learn these costs through experience? Later on, in Chapter 7 we will discuss a method for inferring such costs in a data-driven manner.

Moreover, system authors can also define, implement and use new concept error handling models. A simple example would be the threshold- based model we have mentioned above. More sophisticated models could use additional evidence (e.g. the confidence score of the second best concept hypothesis), and engage additional strategies for recovering from potential misunderstandings (e.g. disambiguation.)

4.4 Request error handling models

If the concept error handling models are in charge of recovering from misunderstandings, the responsibility for handling non-understandings is delegated to the request error handling models. In the RavenClaw error handling architecture, a request error handling model is automatically associated with every request-agent in the dialog task tree. When a non-understanding occurs, the model for the currently focused request agent (i.e. the agent that is on top of the dialog stack) is in charge of engaging an appropriate non-understanding recovery strategy.

In this section, we discuss these models in more detail. We begin by describing the set of available non-understanding recovery strategies in the next subsection, 4.4.1. Then, in subsection 4.4.2, we describe the structure and function of several request error handling models currently available in the RavenClaw dialog management framework.

4.4.1 Recovery strategies

A rich repertoire of non-understanding recovery strategies are available in the RavenClaw dialog management framework. They range from very generic strategies, such as notifying the user that a non-understanding has occurred or asking the user to repeat or rephrase, to strategies that are tailored to specific error conditions such as giving the user examples of how to answer the current system question or telling the user to provide shorter answers or to speak softer. Some of these strategies have parameters which further control their behavior, such as whether or not an apology is issued before the recovery strategy.

Non-understanding recovery strategy	Example
Notify Non-understanding	Sorry, I didn't catch that …
Ask Repeat	Can you please repeat that?
Ask Rephrase	Can you please rephrase that?
Repeat Prompt	Would you like a small room or a large one?
You-Can-Say	For instance, you could say something like "I want a small room", or "I want a large room"
Explain More	Right now I need you to tell me if you would prefer a small room or a large room.
Full Help	I found seven rooms available Friday from 10 to 12. Right now I need you to tell me if you would prefer a small room or a large room. For instance, you could say something like "I want a small room", or "I want a large room"
Interaction Tips	Okay, I know this conversation isn't going well. There are things you can try to help me understand you better. Speak clearly and naturally; don't speak too quickly or too slowly. Give short, concise answers. Calling from a quiet place helps. If you'd like to start from scratch, you can say 'start-over' at any time.
Ask Short Answers and Repeat Prompt	Please use shorter answers because I have trouble understanding long sentences… Would you like a small room or a large one?
Ask Short Answers and You-Can-Say	Please use shorter answers because I have trouble understanding long sentences… For instance, you could say something like "I want a small room", or "I want a large room"
Speak Less Loud and Reprompt	I understand people best when they speak softer. Would you like a small room or a large one?
Move On	Sorry, I didn't catch that. One choice would be Newell Simon 1507. This room can accommodate 50 people, and has a projector, a whiteboard and network access. Do you want a reservation for Newell Simon 1507?
Yield Turn	[…] (system remains silent, yielding the turn to the user)
Ask Start Over	I'm sorry I'm still having trouble understanding you, and I might do better if we restarted. Would you like to start over?
Give Up	I'm sorry but I'm having lots of trouble understanding you and I don't think I will be able to help you. Please call back during normal business hours.

Table 13. Non-understanding recovery strategies in the RavenClaw dialog management framework (suppose a non-understanding happens after the system asks "Would you like a small room or a large one?"

Currently (as of December 2006), 15 such non-understanding recovery strategies are implemented and available for use in the RavenClaw dialog management framework. These strategies are briefly illustrated in Table 13. In the rest of this subsection, we discuss them in more detail (the reader for whom the description in Table 13 is sufficient can safely skip to subsection 4.4.2). Similar to the confirmation strategies used to recover from potential misunderstandings, the non-understanding recovery strategies have been implemented and are available as library dialog agents.

4.4.1.1 Notify Non-understanding

This strategy notifies the user that a non-understanding has occurred – see Figure 40. The strategy is implemented as a library inform agent. When executed, it issues the notification prompt (`{inform notify_nonunderstanding}`), then it requests an Input Pass and completes. In the Olympus framework (see section 3.3 from Chapter 3), a default version of the corresponding prompt for notifying the user (e.g. "Sorry, I didn't catch that…") is already available as part of the set of domain-independent templates in the language generation module. The system author can optionally redefine this prompt.

```
1   SO:    {request date}
    ST:    For which day would you like the conference room?
2   R:     DOES IT AGAIN PLEASE [NONU]
3   SO:    {inform notify_nonunderstanding}
    ST:    Sorry, I didn't catch that ...
```

Figure 40. The Notify Non-understanding recovery strategy (`SO` denotes the semantic output prompt from the dialog manager, and `ST` denotes the corresponding surface form realization; `R` denotes the perceived user response; `[NONU]` marks the non-understanding)

4.4.1.2 Ask Repeat

This strategy asks the user to repeat their previous utterance, which was not understood by the system – see Figure 41. Intuitively, this strategy can be useful if the non-understanding was caused by a transient noise. An additional a `notify` parameter that controls whether or not this strategy first notifies the user that a non-understanding has occurred – see the second example in Figure 41. Unless otherwise mentioned, this parameter is available for all the other strategies described in the rest of this section.

The `Ask Repeat` strategy is implemented as a library inform-agent, in a manner similar to the `Notify Non-understanding` version. Predefined versions of the corresponding prompts (both for asking the user to repeat and for notifying that a non-understanding has occurred) are already

Example 1

```
1   SO:    {request date}
    ST:    For which day would you like the conference room?
2   R:     DOES IT AGAIN PLEASE [NONU]
3   SO:    {inform ask_repeat}
    ST:    Could you please repeat what you said?
```

Example 2

```
1   SO:    {request date}
    ST:    For which day would you like the conference room?
2   R:     DOES IT AGAIN PLEASE [NONU]
3   SO:    {inform notify_nonunderstanding} {inform ask_repeat}
    ST:    Sorry, I didn't catch that … could you please repeat
           what you said?
```

Figure 41. The Ask Repeat non-understanding recovery strategy (example 1 is the default behavior, example 2 includes a notification that a non-understanding has occurred)

available as part of the domain-independent templates in the language generation module, and can be overwritten by the system author.

4.4.1.3 Ask Rephrase

This third strategy is similar to the previous one. Instead of asking the user to repeat, this strategy asks the user to rephrase their previous utterance – see Figure 42. Intuitively, this strategy can be useful if the non-understanding is caused by an input that falls outside the system's grammar or language modeling abilities.

```
1   SO:    {request date}
    ST:    For which day would you like the conference room?
2   R:     DOES IT AGAIN PLEASE [NONU]
3   SO:    {inform ask_rephrase}
    ST:    Could you please rephrase that?
```

Figure 42. The Ask Rephrase non-understanding recovery strategy

From an implementation standpoint, the strategy is also very similar to the `Ask Repeat` strategy. Predefined versions for the system prompts (both for asking the user to rephrase and for notifying that a non-understanding has occurred) are available, and can be overwritten by the system author.

4.4.1.4 Repeat Prompt

In the `Repeat Prompt` strategy, the system repeats its previous prompt (question) – see Figure 43. This strategy can be useful in turn overtaking situations, i.e. if the user barges-in on the system and as a result does not hear the last system prompt, or in any other situation when the user has not correctly understood the last system question.

The strategy is implemented as an inform-agent that reissues the prompt corresponding to the request-agent that was in focus at the time the non-understanding occurred. The strategy has access to this prompt since upon instantiation the error handling decision process passes it a pointer to the request-agent in focus.

```
1   SO:    {request date}
    ST:    For which day would you like the conference room?
2   R:     DOES IT AGAIN PLEASE [NONU]
3   SO:    {request date}
    ST:    For which day would you like the conference room?
```

Figure 43. The Repeat Prompt non-understanding recovery strategy

4.4.1.5 You-Can-Say

The `You-Can-Say` strategy gives the user a few examples of how to answer the current system question – see Figure 44. In general, this strategy can be helpful when the user's response falls outside the system's grammar or language model.

The `You-Can-Say` strategy is implemented as an inform-agent that issues the `what_can_i_say` version of the prompt (see section 3.2.2.2 from the previous chapter) corresponding to the request-agent that was in focus when the non-understanding occurred. When using this strategy, system authors must define the `what_can_i_say` version of the prompts for each request-agent in the dialog task tree (this is in fact already required to support the user-triggered `Help` conversational strategy discussed in section 3.2.4.1 from the previuos chapter).

```
1  SO:   {request date}
   ST:   For which day would you like the conference room?
2  R:    DOES IT AGAIN PLEASE [NONU]
3  SO:   {request date version=what_can_i_say}
   ST:   For instance, you could say something like 'Monday'
         or 'tomorrow' or 'next Tuesday'
```

Figure 44. The You-Can-Say non-understanding recovery strategy

4.4.1.6 Explain More

The `Explain More` strategy gives a more comprehensive version of the previous system prompt – see for instance Figure 45. Like the `Repeat Prompt` strategy, the `Explain More` strategy can be useful in situations when the user has not correctly or fully understood the last system question.

```
1  SO:   {request date}
   ST:   For which day would you like the conference room?
2  R:    DOES IT AGAIN PLEASE [NONU]
3  SO:   {request date version=explain_more}
   ST:   Right now I need you to tell me for which day you
         would like to reserve the conference room
```

Figure 45. The Explain More non-understanding recovery strategy

The strategy is implemented as an inform-agent that issues the `explain_more` version of the prompt for the request-agent that was in focus at the time the non-understanding occurred. The system author must define these prompts for all request-agents in the dialog task tree.

4.4.1.7 Full Help

The `Full Help` strategy provides a comprehensive help message that contains a description of the current system state, the more comprehensive version of the previous system prompt and a few examples of appropriate answers at this point in the dialog – see Figure 46.

The strategy is implemented as an inform-agent which issues in turn the `establish_context`, `explain_more`, and `what_can_i_say` versions of the prompt for the request-agent that was in focus at the time the non-understanding occurred. The strategy is very similar to the user-triggered `Help.FullHelp` conversational strategy described in section 3.2.4.1 from the previuos chapter. The system author must define these prompts for all request-agents in the dialog task tree (this is already required to support the user-triggered `Help.FullHelp` conversational strategy.)

```
1  SO:   {request date}
   ST:   For which day would you like the conference room?
2  R:    DOES IT AGAIN PLEASE [NONU]
3  SO:   {request date version=establish_context}
         {request date version=explain_more}
         {request date version=what_can_i_say}
   ST:   I am currently trying to gather enough information
         to make a room reservation for you. So far I know
         you want the room at 10 a.m. Right now I need you to
         tell me for which day you would like to reserve the
         conference room. For instance, you could say
         something like 'Monday' or 'tomorrow' or 'next
         Tuesday'
```

Figure 46. The Full Help non-understanding recovery strategy

4.4.1.8 Interaction Tips

The `Interaction Tips` recovery strategy gives provides a help message containing generic guidelines for how to best interact with the system – see Figure 47. The strategy is similar to the user-triggered `Help.InteractionTips` conversational strategy described in 3.2.4.1 from the previuos chapter.

```
1  SO:   {request date}
   ST:   For which day would you like the conference room?
2  R:    DOES IT AGAIN PLEASE [NONU]
3  SO:   {inform interaction_tips}
   ST:   Okay, I know this conversation isn't going well.
         There are things you can try to help me understand
         you better. Speak clearly and naturally; don't speak
         too quickly or too slowly. Give short, concise
         answers. Calling from a quiet place helps. If you'd
         like to start from scratch, you can say `start-over'
         at any time.
```

Figure 47. The Interaction Tips non-understanding recovery strategy

The `Interaction Tips` strategy is implemented as an inform-agent. A predefined corresponding prompt is available as part of the set of domain-independent language generation templates; this prompt can be redefined by the system author.

4.4.1.9 Ask Short Answers and Repeat Prompt

This strategy asks the user to provide a short answer, and then reissues the previous system prompt – see Figure 48. The strategy is targeted for situations when the user response is either too long, or ungrammatical.

```
1  SO:   {request date}
   ST:   For which day would you like the conference room?
2  R:    DOES IT AGAIN PLEASE [NONU]
3  SO:   {inform ask_short_answer} {request date}
   ST:   Please use shorter answers because I have trouble
         understanding long sentences. For which day would
         you like the conference room?
```

Figure 48. The Ask Short Answers and Repeat Prompt non-understanding recovery strategy

The strategy is implemented as a library inform agent that first issues a generic prompt asking the user to speak shorter utterances, and then reissues the prompt corresponding to the request-agent that was in focus when the non-understanding occurred. A predefined version of the prompt asking the user to speak shorter is available; system authors can redefine this prompt.

4.4.1.10 Ask Short Answers and You-Can-Say

This strategy asks the user to provide a short answer, and then gives a few sample answers to the current system question – see Figure 49. In general, this strategy can be useful when the user response is too long, ungrammatical, and/or does not match the system expectations. Providing information about possible responses at the current point in the dialog can shape user behavior and help recover from the non-understanding.

The strategy is implemented as a library inform-agent that first issues a prompt asking for a short answer, and then issues the `what_can_i_say` version of the prompt for the request-agent that was in focus at the time the non-understanding occurred. A predefined version of the prompt asking the user to speak shorter utterances is available, and can be overwritten by system authors. When

```
1  SO:    {request date}
   ST:    For which day would you like the conference room?
2  R:     DOES IT AGAIN PLEASE [NONU]
3  SO:    {inform ask_short_answer} {request date
           version=what_can_i_say}
   ST:    Please use shorter answers because I have trouble
          understanding long sentences. For instance, you
          could say something like 'Monday' or 'tomorrow' or
          'next Tuesday'
```

Figure 49. The Ask Short Answers and You-Can-Say non-understanding recovery strategy

using this strategy, the `what_can_i_say` prompts for all request-agents in the dialog task tree need to be defined at the language generation level.

4.4.1.11 Speak Less Loud and Repeat Prompt

This strategy asks the user to speak less loud, and then reissues the previous system prompt – see Figure 50. The strategy is useful when the user is speaking too loud, for instance because of accumulated frustration, adverse noise conditions, etc. Loud speech can lead to clipping which in turn leads to significant degradation of recognition performance.

The `Speak Less Loud and Repeat Prompt` strategy is implemented as a library inform-agent that first asks the user to speak less loud, and then reissues the prompt corresponding to the request-agent that was in focus when the non-understanding occurred. A predefined version of the prompt asking the user to speak less loud is available; system authors can redefine this prompt.

```
1  SO:    {request date}
   ST:    For which day would you like the conference room?
2  R:     DOES IT AGAIN PLEASE [NONU]
3  SO:    {inform ask_less_loud} {request date}
   ST:    I understand people best when they speak softer. For
          which day would you like the conference room?
```

Figure 50. The Speak Less Loud and Repeat Prompt non-understanding recovery strategy

4.4.1.12 Move On

The `Move On` strategy is different from the previously described strategies in that it does not try to directly address the current non-understanding. Instead, the system ignores the current non-understanding and continues the dialog by moving on to a different question, or by switching to an alternative dialog plan for accomplishing the same goal. Figure 51 provides two examples. For instance, in the first example (extracted from the RoomLine system) the system abandoned the goal of finding out whether the user would like a small or a large room, and started suggesting rooms from the current list. In the second example (extracted from the Let's Go! Public system), the system abandoned the goal of directly identifying the departure stop and moved on to first trying to identify the neighborhood.

Typically, as the number of consecutive errors increases in a conversation, so does user frustration [39, 100]. This generally leads to hyper-articulation and other deviations from normal speech patterns, which in turn further exacerbate the recognition difficulties, leading to more errors. A vicious circle of errors (as known as an error spiral) is created. When using the `Move On` strategy, the system ignores the current non-understanding and continues with a different question. This can alleviate user frustration and break the spiral of errors. Skantze has shown in a wizard-of-oz study that this strategy is very often used by human operators faced with the same type of understanding-errors as a dialog system [115].

Example 1

```
1  SO:    {request room_size}
   ST:    I found 4 rooms Friday between 1 and 3 p.m. Would
          you like a small room or a large one?
2  R:     ANYTIME CECILY [NONU]
3  SO:    {inform suggest_room <results suggest=1}
          {request want_this_room}
   ST:    One choice would be Newell Simon 1507. This room can
          accommodate 50 people, and has a projector, a
          whiteboard and network access. Do you want a
          reservation for Newell Simon 1507?
```

Example 2

```
1  SO:    {request departure_stop}
   ST:    Where are you leaving from?
2  R:     OKAY IN THAT SAME PAY [NONU]
3  SO:    {inform step_by_step} {request neighborhood}
   ST:    Let's proceed step by step … Which neighborhood
          would you like to leave from?
```

Figure 51. The Move On non-understanding recovery strategy (example 1 is extracted from the RoomLine system; example 2 is extracted from the Let's Go! Public system)

The `Move On` strategy is implemented by means of a library execute-agent that sets the request-agent that was in focus to a "completed-with-failure" state (see section 3.2.2.2 from Chapter 3.) The system author needs to ensure that the dialog control logic successfully handles the cases when the request agents that have the `Move On` strategy available fail.

4.4.1.13 Yield Turn

In the `Yield Turn` strategy, the system remains silent after a non-understanding, pretending in effect that it never heard the user. Sometimes, users will repeat their last utterance in this case.

The strategy is implemented as an inform agent that issues an empty prompt. The `notify` parameter previously described is not available for this strategy.

4.4.1.14 Ask Start Over

The `Ask Start Over` strategy notifies the user that the system is having problems understanding correctly, and asks whether the user would like to start-over from the beginning. The strategy is illustrated in Figure 52: in the first example, the user declines; in the second example the user complies. Like the `Move On` strategy, this strategy also aims to break a spiral of errors by giving the user an opportunity to eliminate in one turn all the potential misunderstandings the system has accumulated and start afresh.

The strategy is implemented via a library dialog agency consisting of (1) a request-agent that asks the user whether they would like to restart the conversation, (2) an inform-agent that notifies the user that the conversation will be restarted, and (3) an execute-agent that reinitializes the dialog (the last two agents are preconditioned on a positive user response to the "Would you like to start over" question). Default versions of the system prompts corresponding to the request- and inform-agents are available as part of the domain-independent language generation templates, and can be redefined by the system author.

Example 1

```
1  SO:   {request date}
   ST:   For which day would you like the conference room?
2  R:    DOES IT AGAIN PLEASE [NONU]
3  SO:   {request nonu_want_start_over}
   ST:   I'm having lots of trouble understanding you, but I
         might do better if we restarted. Would you like to
         start over?
4  R:    NO
5  SO:   {request date}
   ST:   For which day would you like the conference room?
```

Example 2

```
1  SO:   {request date}
   ST:   For which day would you like the conference room?
2  R:    DOES IT AGAIN PLEASE [NONU]
3  SO:   {request nonu_want_start_over}
   ST:   I'm having lots of trouble understanding you, but I
         might do better if we restarted. Would you like to
         start over?
4  R:    YES
5  SO:   {inform starting_over} {inform welcome} ...
   ST:   Okay, let's restart. By the way, you can say 'help'
         at any time if you get stuck. Welcome to
         Roomline ...
```

Figure 52. The Ask Start Over non-understanding recovery strategy (in example 1 the user declines, in example 2 the user accepts)

4.4.1.15 Give Up

The `Give Up` strategy terminates the conversation and asks the user to call at a later time – see Figure 53. The strategy can be useful if the conditions are such as the likelihood of success in the conversation is very small, for instance adverse noise conditions, very poor recognition, etc. In this case the system might be better off terminating the conversation or directing the user to a human operator. The system is then free to serve another call.

```
1  SO:   {request departure_stop}
   ST:   Where are you leaving from?
2  R:    OKAY IN THAT SAME PAY [NONU]
3  SO:   {inform give_up}
   ST:   I'm sorry but I'm having lots of trouble
         understanding you and I don't think I will be able
         to help you. Please call back during normal business
         hours.
```

Figure 53. The Give Up non-understanding recovery strategy

The strategy is implemented as a library inform-agent that first issues a prompt informing the user the conversation will be terminated, and then sends an end-of-session signal to the dialog engine. A predefined version of the corresponding prompt is available as part of the domain-independent language generation templates; this prompt can be redefined by the system author.

4.4.2 Request-agent error handling models

After having discussed the repertoire of non-understanding recovery strategy available in the RavenClaw dialog management framework, we now turn our attention to the request-agent error handling models which are in charge of engaging these strategies. If concept-level error handling models are

associated with every concept in the dialog task specification, similarly, request-agent error handling models are associated with every request agent in the tree. When a non-understanding occurs, the model for the currently focused agent is in charge of engaging an appropriate non-understanding recovery strategy. The request-agent error handling models therefore implement control policies over the set of available non-understanding recovery strategies.

We have seen that for misunderstandings detection is a key issue, and the set of confirmation strategies that can be engaged is relatively small and well understood. In contrast, for non-understandings, the set of strategies that could be used to recover is much larger – see the 15 strategies described in the previous subsection. The relative trade-offs between these strategies are less obvious; in fact, they might be task- and domain-specific. Intuitively, certain strategies are more helpful in certain situations. For instance, asking the user to repeat might be an appropriate course of action if the non-understanding was caused by a transient noise. In contrast, if the non-understanding is the result of an out-of-vocabulary word, asking the user to repeat will not be very helpful. Correctly diagnosing the source of a non-understanding is also a difficult problem. As a consequence, designing a good control policy over these strategies is in general a challenging task.

Typically, most spoken dialog system use a limited set of non-understanding recovery strategies and very simple heuristic policies for engaging them. A typical example is the so-called "three strikes and you're out" approach [5]: repeat the system question after the first non-understanding, provide more help after the second one, and transfer the user to a human operator if a third consecutive non-understanding occurs.

RavenClaw supports this type of heuristic policies, via two different types of request-agent error handling models: `default` and `NoN` (number-of-non-understandings). We describe these models in more detail in the next two subsections – 4.4.2.1 and 4.4.2.2. Other heuristic-based error handling models can be easily defined and used by a system author. We argue however that these models are ad-hoc in nature, and do not accurately capture the trade-offs, costs and particular characteristics of the domain in which the system operates. To address this issue, we have developed an online, supervised learning based approach for deriving non-understanding recovery policies from data. The method and corresponding empirical results are described in detail in section 8.4 from Chapter 8. Here, we limit ourselves to outlining the structure of the corresponding error handling model in subsection 4.4.2.3.

4.4.2.1 The `default` request-agent error handling model

Like for the concept error handling models, the representation for the `default` request-agent error handling model is a Markov decision process (MDP).

The structure of this model is illustrated in Figure 55. The model has 2 states: `understanding`, `non-understanding`. In contrast to the concept error handling model, the request model is fully observable since non-understandings are automatically detected by the system.

The only available action from the `understanding` state is `no-action` (this will be engaged by the model at each time step when no non-understanding occurs). The set of non-understanding recovery strategies described previously are available and can be engaged from the `non-understanding` state.

The policies for this model are defined in a similar manner to the concept-level policies, i.e. by specifying the relative utilities for various strategies. The `exploration_mode` and `exploration_param` parameters are also available for the request-agent policies. In addition, system authors can further customize this error handling model by overwriting a method that defines which strategies are available from the `non-understanding` state. System authors can therefore use any other information available at runtime in the system to constrain which strategies are available at any given point. For instance, rules such as "don't ask the user to repeat three times in a row", or "don't ask for a short answer if the current non-understanding has a small number of words already", etc. can be enforced via this method. At each decision point, the model chooses between the available strategies based on their defined utilities and specified the exploration policy. The `default` request-agent error

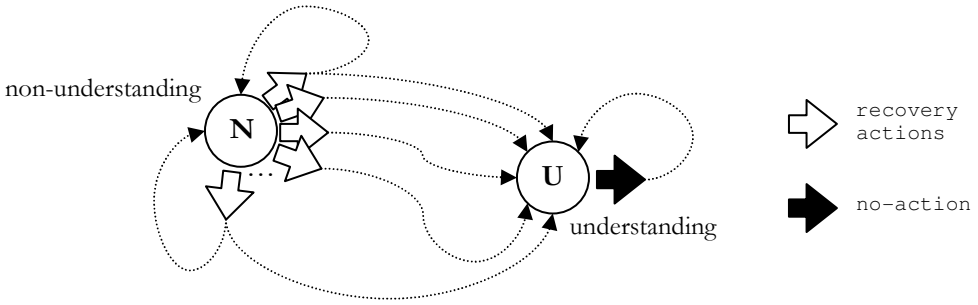


Figure 54. Structure for the default request-agent error handling model (not all transitions are illustrated)

handling model therefore allows system authors to create stochastic heuristic policies.

4.4.2.2 The NoN request-agent error handling model

The NoN (Number-of Non-understandings) request-agent error handling model is very similar to the default model described in the previous subsection. This model is implemented as an MDP that uses information about the number of consecutive non-understandings in the current error segment. The model consists of four states, illustrated in Figure 55: nonu_1, nonu_2, nonu_3_or_more and understanding. The model is in the state nonu_1 when the current non-understanding is the first one in a segment, i.e. when the previous utterance was not a non-understanding. The model is in the state nonu_2 when this is the second non-understanding in a segment, i.e. when the previous utterance was a non-understanding but the one before that was not. Finally, the model is in the state nonu_3_or_more when there were at least two previous non-understandings preceding the current one. The non-understanding recovery strategies are available as actions from the 3 non-understanding states, and only no-action is available from the understanding state. This model structure allows the system author to specify different utilities for the different non-understanding recovery strategies, depending on whether or not this is the first, second or later non-understanding in a non-understanding segment. Like for the default model, system authors can overwrite a routine that defines which strategies are available at any given point in the dialog.

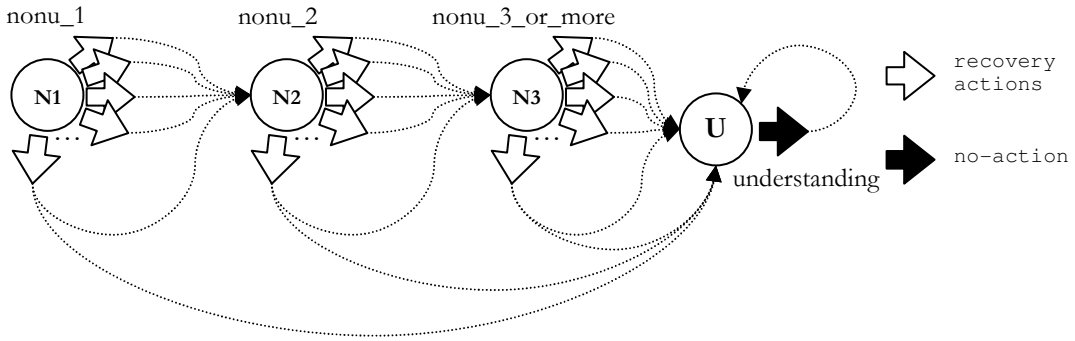


Figure 55. Structure for the NON request-agent error handling model (not all transitions are illustrated)

4.4.2.3 The LR (data-driven) request-agent error handling model

The default and NoN request-agent error handling models described in the previous two sections allow a system author to implement various stochastic heuristic policies. This is an ad-hoc approach that relies on rules of thumb and expert knowledge. It is typically used in most spoken language interfaces. In addition to these models, RavenClaw also supports a data-driven request-agent error handling model.

We start from the intuition that certain non-understanding recovery strategies are more likely to succeed under certain circumstances. For instance, if the source of the non-understanding is an out-of-vocabulary word, asking the user to repeat is less likely to help than asking the user to rephrase. However, if the non-understanding is caused by a transient noise, then asking the user to repeat might be a more appropriate course of action. If we could accurately estimate the likelihood of success for each non-understanding recovery strategy (success being defined as "follow-up user response will be correctly understood by the system"), the optimal policy would be relatively easy to construct: simply select the strategy with the highest likelihood of success.

The data-driven request-agent error handling model relies on a number of such predictors – one for each non-understanding recovery strategy. The individual predictors are trained in a supervised learning fashion, using logistic regression models (hence the name `LR` for this error handling model), and a large set of features available online to the system (the approach is described in detail in section 8.4 from Chapter 8.) At runtime, the system uses these predictive models to estimate the likelihood of success for each non-understanding recovery strategy given the current situation. In essence, we are using logistic regression to model the probability of transitioning to the `understanding` state from the `non-understanding` state when a certain action is taken, $P(U|N, A_k)$ in the MDP from Figure 54. Based on the constructed estimates, the model then chooses the strategy that is most likely to succeed.

Additionally, the individual logistic regression models provide confidence intervals for each predicted likelihood of success. These confidence intervals reflect the system's certainty about that prediction and allow us to guide the system in an online approach to learning the individual models. The system starts with a set of agnostic models, and, as more data is gathered the individual models are retrained and become more and more accurate; the system learns a non-understanding recovery policy online. In section 8.4 from Chapter 8, we discuss in more detail the technical aspects of this learning-based approach and present empirical results in a deployed spoken dialog system.

4.5 Summary and future directions

In this chapter we have described the error handling architecture in the RavenClaw dialog management framework. The architecture is task-independent and decouples both the error handling strategies and the error handling decision process from the actual dialog task specification. The error handling strategies are implemented as library dialog agencies, using the RavenClaw dialog task specification formalism (see previous chapter.) The error handling decision process is implemented in a distributed fashion, as a collection of local, independent error handling models associated with every concept and request-agent in the dialog task specification. The concept error handling models are in charge of handling potential misunderstandings: they use information about the confidence score of each heard concept value, and can engage various confirmation actions. The request error handling models are in charge of recovering from non-understandings: they can trigger a wide array of non-understanding recovery strategies, such as asking the user to repeat, asking the user to rephrase, providing help, etc. While task-decoupled implementations of various conversational components have been advocated before in different contexts [5, 65, 71, 99, 122, 123], to our knowledge this is the first task-independent error handling architecture implemented in a complex, plan-based dialog management framework.

This task-decoupled solution provides a number of advantages. First, it lessens the system development effort. System authors specify at a high level which recovery strategies and policies should be used, and focus on developing the domain-specific dialog control logic under the assumption that inputs to the system will always be perfect. The responsibility for ensuring that the conversation is on track and advances normally is delegated to the underlying error handling architecture. The architecture is modular and promotes the reusability of components: error indicators, strategies and policies can be developed separately can be easily plugged into any new or existing RavenClaw-based system. In addition, the task decoupled approach ensures consistency and uniformity in behavior both within and across systems. It also represents a more sensible solution from a software engi-

neering perspective.

The distributed nature of the error handling process represents a strength but perhaps also a weakness. On one hand, this approach confers good scalability properties, supports dynamic generation of dialog tasks[14], and favors learning based approaches for error handling. The error handling problems are encapsulated (and solved) locally and decoupled from the actual task performed by the system. Parameters for different error handling models can be tied across models; this in turn improves scalability of learning-based approaches, because the total number of parameters to be learned grows sublinearly with the size of the dialog task. At the same time, the error handling models operate locally, on individual concepts and requests in the dialog plan. This raises a number of questions. For instance, long-range dependencies (e.g. inter-concept relationships) are not directly captured; in some situations, these dependencies can provide additional useful information for the error handling process. For instance, knowing that the start-time for a room reservation is 2p.m. puts a number of constraints on what the end-time can be. Furthermore, the error handling strategies operate individually on concepts or request agents. The gating mechanism implements a heuristic that can sometimes couple multiple recovery actions in a single turn. For instance an implicit confirmation can be coupled with a follow-up explicit confirmation on a different concept: "a small room … Did you say in Wean Hall?". Difficulties arise however if we want the system to combine multiple concepts in a single confirmation action, such as "Did you say you wanted a small room in Wean Hall?". The view taken in this work is that, to a large extent, error handling can be modeled as a local process. I believe that significant gains in overall performance can be obtained by optimizing the error handling behaviors locally. Nevertheless, in future work it would be interesting to consider and address the limitations described above.

Together with the larger RavenClaw dialog management framework and the system built on top of it, the error handling architecture we have presented in this chapter provides the infrastructure and experimental platform for the rest of the work described in this dissertation. Throughout this chapter, we have already pointed to a number of important and challenging questions: how can we accurately detect misunderstandings? How can a system update its beliefs based on evidence collected throughout a conversation? How can we set various confidence thresholds? How can a system learn a non-understanding recovery policy over a large set of strategies from data? In the remaining chapters we address these, as well as a few other issues. We begin from the issue of detecting misunderstandings.

[14] The dialog task tree can grow dynamically at runtime; the corresponding concept and request-agent error handling models will be automatically created by the dialog engine.

PART III.

HANDLING MISUNDERSTANDINGS

Chapter 5

Confidence annotation

In order to detect potential misunderstandings, spoken language interfaces typically rely on recognition confidence scores. In this chapter we investigate various learning methodologies for building semantic confidence annotation models. We begin by focusing on supervised learning techniques. We report on a comparative analysis of four such techniques (i.e. logistic regression, boosting, decision trees and Naïve Bayes), using corpora from three different spoken dialog systems. We investigate different evaluation metrics, the advantages and disadvantages of various supervised learning models, the relationship between training set size and performance, and to which extent confidence annotation models generalize across domains. Next, in the second part of this chapter, we propose a novel, implicitly supervised approach for learning confidence annotation models. This approach eliminates the need for developer supervision; instead, the system obtains its supervision signal directly from the interaction, from user responses to system confirmation strategies. We believe this implicitly-supervised learning approach is applicable to other problems, and represents an important step towards developing autonomous, self-improving interactive systems.

5.1 Introduction

In the introductory chapter, we have identified two types of understanding-errors that commonly affect spoken language interfaces: misunderstandings and non-understandings. In this chapter, we focus our attention on misunderstandings, more specifically on the problem of detecting these errors.

By definition, we say that a misunderstanding happens when the system constructs an incorrect discourse-level interpretation of the user's turn. Figure 56 provides an example. In turn 2, the user responded *"I need a reservation for next Thursday"*; unfortunately, due to speech recognition errors, this answer is understood as *"I need a reservation for next Tuesday"*. In order to detect such misunderstandings, most spoken dialog systems rely on **confidence scores**. These scores are automatically computed by the system at runtime, and they reflect the likelihood that the decoded result corresponds to the user's expressed intention. Confidence scores can be constructed at the lexical or semantic level, and can be associated with words, concepts, or entire utterances. For instance, in the example from Figure 7, the utterance-level confidence score was 0.68: the system believes that, with probability 0.68, the user's expressed intent in this utterance was to reserve a room for next Tuesday. The problem of computing these scores is known as the **confidence annotation**

1 S: Welcome to RoomLine, the automated conference room reservation system. How may I help you?
2 U: *I need a reservation for next Thursday*
R: I NEED A RESERVATION FOR NEXT TUESDAY / 0.68
P: [date=next Tuesday]

Figure 56. Example misunderstanding in a conference room reservation system (S: marks the system turns, U: marks the user turns, R: marks the recognition result, P: marks the semantic representation of the recognition result)

problem. In the context of conversational spoken language interfaces, detecting misunderstandings reduces therefore to performing accurate semantic-level confidence annotation.

The confidence annotation problem has already received a fair amount of attention. Traditionally, work on confidence annotation was conducted in the context of dictation and broadcast news transcription tasks [30, 49, 112, 118, 133]. Accordingly, the proposed schemes work at the lexical level. For instance, word-level confidence annotation models assign a reliability tag (or a probability score) to each word token in the decoder hypothesis. However, in the context of spoken dialog systems, the high-level semantic interpretation is what really matters: small lexical errors can be safely absorbed by a dialog system because the dialog manager actions are dictated by the input semantics. As a result, a number of researchers [18, 38, 44, 52, 53, 70, 85, 102, 104, 117], including the author [9, 22], have investigated various methods for building confidence annotators operating at the semantic level. Typically, the problem is cast as a supervised learning task: given a set of features that characterize the current semantic hypothesis and the current context, does this semantic hypothesis correspond to the user's expressed intention?

Although the semantic confidence annotation problem has also received a significant amount of attention (see also section 5.2.), a number of interesting questions remain to be explored:

(4) **what is an appropriate metric for evaluating confidence annotation performance in the context of a conversational spoken language interface?** Typically, confidence annotators are evaluated in terms of classification error, accuracy, or ROC curves. These metrics are well suited when one is interested in making accept/reject decisions based on the confidence score. They are however not appropriate if one plans to use the confidence score as a probability estimate, for instance to make more fine-grained confirmation decisions. We discuss this issue in more detail in section 5.3. We propose two alternative metrics, and empirically show that an evaluation based only on classification error can create misleading results (see subection 5.4.2.5.)

(5) **what are the advantages and disadvantages of various supervised learning techniques on this task?** In previous work [22], we have comparatively evaluated six different classification models on the semantic confidence annotation task. With the notable exception of the Naïve Bayes classifier (which performed somewhat worse), no statistically significant differences were found between the other models. Since then, a few other authors have compared different supervised learning techniques for this task. Their results generally corroborate our initial analysis. In [118] Stemmer reports no significant differences between decision trees and artificial neural networks. In [116] Skantze reports no significant differences between a memory-based and a transformation-based learner. In [75] Moreno reports very small improvements of a boosting-based algorithm over decision trees and support-vector-machines. In this chapter we extend this body of work by performing a more detailed comparative evaluation of four supervised learning techniques on three different corpora.

Current supervised learning techniques rely on the existence of a labeled corpus from which the confidence annotation model parameters are estimated. Unfortunately labeled corpora are expensive and difficult to acquire, especially in the early stages of system development and deployment.

Furthermore, supervised learning favors an off-line, batch approach that leads to fixed models that do not respond well to changes in the system's environment. In an effort to address these limitations, we have also investigated the following questions:

(6) **what is the relationship between training set size and confidence annotation performance?** How much data is enough? Are some supervised learning techniques more sample efficient than others?

(7) **can we successfully transfer a confidence annotator that was trained with data from domain A into domain B without any labeled data from B?** Alternatively, can we adapt the annotator into the new domain with a minimal amount of labeling effort?

(8) **can we train a confidence annotator in an unsupervised fashion, or bootstrap it from small amounts of labeled data?** Can a system learn and adapt its confidence annotation model online, without explicit supervision from its developers?

In this chapter, we discuss in detail the problem of semantic confidence annotation in conversational spoken language interfaces, and address each of the questions raised above. We begin with a brief review of related work in section 5.2. Then, in section 5.3, we formalize the semantic confidence annotation problem and discuss different evaluation methodologies (question 1). In section 5.4 we discuss a number of issues related to the supervised learning approach for training confidence annotators. We conduct experiments with multiple corpora and compare four supervised learning techniques on the confidence annotation task (question 2). We also empirically investigate the relationship between the amount of available training data and annotator performance (question 3), and the portability of the constructed confidence annotators across different domains (question 4). Finally, in section 5.5 we propose and evaluate a novel, implicitly supervised approach for building semantic confidence annotation models (question 5). We conclude this chapter in section 5.6 by outlining some potential directions for further extending this work.

5.2 Related work

5.2.1 Lexical confidence annotation

Initially, confidence annotation schemes have been developed in the context of speech recognition tasks such as dictation or broadcast news transcription [30, 49, 112, 118, 133]. The proposed schemes use language and acoustic model information, as well as search information captured in the recognition lattices or n-best lists. They assign a confidence score either to the whole recognized hypothesis [49] or to each word in the recognized hypothesis [30, 49, 112, 118, 133]. Here are some examples. Hazen et al [49] use normalized acoustic scores at the phone-level, word-level and utterance-level in conjunction with a discriminative training procedure to derive phone-level, word-level and utterance-level confidence scores. Wessel et al [133] describe a methodology for constructing word-posterior probabilities (i.e. word-level confidence scores) by using information available in the recognition lattice or in the n-best list. Others have proposed integrating additional features in the confidence annotation process. Shi and Zhou [112] report improvements in a word-level confidence annotation task for dictation by using part-of-speech information as well as syntactic features derived from a link grammar. Similarly, Stemmer et al [118] use an artificial neural network to combine different word-level features (e.g. part-of-speech, length, frequency) with acoustic measures and word-graph based features. Cox and Dasmahapatra [30] use latent semantic analysis to construct between-words semantic similarity scores, and then use this information to construct a word-level confidence score. They showed that, when combined with an n-best list based confidence annotator, this technique can produce improved results over the simple n-best list based approach.

5.2.2 Semantic confidence annotation

The confidence annotation schemes described above operate at the lexical level. However, in the context of spoken dialog systems, actions are determined by the semantic representation of the recognized result. Lexical errors do not matter if the semantics of the utterance are correctly captured. For instance, if the user responds *"yes"* but the recognizer produces *"yeah"*, although the have 100% word-error-rate, the semantic-error-rate is 0%. An accurate (low in this case) recognition confidence score would not be helpful for a spoken dialog system. In general, although word-error-rate and concept-error-rate are correlated, this correlation is not perfect. In Figure 57 we show the empirical correlation between these metrics in three different dialog systems; the correlation coefficients range from 0.59 to 0.76.

Given this incongruence, the focus changes in the context of spoken dialog systems, from generating confidence scores that reflect the lexical accuracy of the decoded hypothesis, to generating confidence scores that reflect the semantic accuracy. In the sequel, we will use the term recognition confidence scores to denote the first category, and semantic confidence (or simply confidence scores) scores to denote the latter.

We have already seen that additional features such as part-of-speech information [112, 118] and semantic similarity [30] can provide useful information for constructing recognition confidence scores. The same holds true for semantic confidence scores. In spoken dialog systems, the user input typically passes through at least two additional processing stages: language understanding and discourse interpretation. These stages can provide additional information for the semantic confidence annotation task. For instance, utterances that do not parse well, or utterances that do not match the dialog manager expectations are more likely to be misunderstandings. As a result, a number of researchers [18, 38, 44, 52, 53, 70, 85, 102, 104, 117], including the author [9, 22], have proposed integrating information from these various knowledge sources to derive more accurate semantic-level confidence scores.

For instance, in [22] we have assessed the utility of several speech recognition, language understanding and dialog-level features for constructing a semantic confidence annotator for the CMU Communicator, a spoken dialog system operating in the air travel domain [101]. We used a corpus of 4550 utterances collected with this system, and investigated six different supervised learning techniques: Bayesian networks, boosting, decision trees, neural networks, support vector machines and a Naïve Bayes classifier. The results indicate that all classifiers except for Naïve Bayes attained a classification error rate of about 18%. This result was equivalent with a 45% relative reduction in error from a majority baseline, and a 30% relative reduction in error from the heuristic rules previously used for confidence annotation in the CMU Communicator system.

In a similar study [70], Litman et al used RIPPER, a rule-based machine learning approach to derive a semantic confidence annotator. The data used in their study came from three different spo-

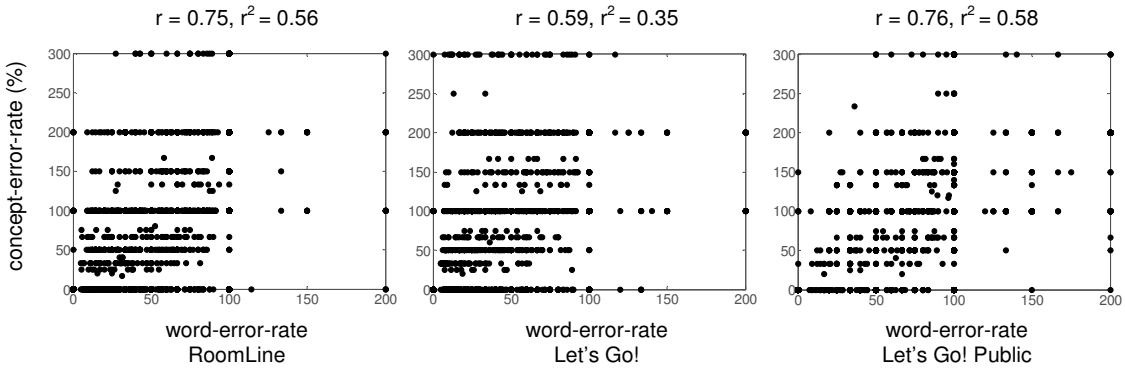


Figure 57. Relationship between turn-level word- and concept-error-rate in three different dialog domains

ken dialog systems: ANNIE – an agent for voice dialing and messaging, ELVIS – an agent for accessing email, and TOOT – an agent for accessing online train schedules. The authors used a large number of features from different knowledge sources in the system: acoustic features, dialog efficiency features, dialog quality features, lexical features, etc. The trained confidence model attained a classification error rate of 23%, equivalent with a 52% relative reduction from the majority baseline (which was 48% in their case).

These two studies are representative for a larger body of work on semantic confidence annotation [9, 18, 38, 44, 52, 53, 85, 102, 104, 117]. Typically, the problem is cast as a supervised learning task: given a set of features characterizing the current recognition result and the current context, predict whether or not the user's utterance was correctly understood by the system. The general methodology is as follows.

First, a corpus of in-domain user utterances is collected and each utterance is manually labeled as either correctly-understood or misunderstood by the system. In some approaches [44, 102, 104], the labeling and the subsequent confidence annotation are performed at the concept rather than utterance level. The training corpora can range in size from hundreds [38, 52] to tens of thousands of utterances [44, 85].

Second, a set of potentially relevant features from different knowledge sources in the system are identified. Various features have been proposed and evaluated to date: decoder level features (acoustic and language model scores, lattice information) [18, 38, 70, 85, 102, 117], other acoustic information such as energy, pauses and prosodic characteristics [70], lexical information [53, 117] (e.g. presence or absence of certain word-tokens), part-of-speech information [117], parsing scores and other grammar-level features [22, 85], discourse-level features [18, 22, 52, 53, 70], as well as pragmatic plausibility features [38]. Because the confidence annotation models need to run live in the system, only features that can be computed at run-time are used.

The third component is the supervised learning method. A large variety of such methods have been explored to date: artificial neural networks [44, 102], decision trees [85, 104], support-vector-machines [75, 117], transformation-based-learning [112], linear discriminant functions [52], rule-based learning [18, 38, 53, 70], memory-based learning [18, 38, 116], and boosting [75].

In the final step, the confidence annotator is trained using a portion of the collected corpus and evaluated on a held-out test set. Typically, the evaluation is performed by computing the classification error and comparing it against majority (always predicting the majority class) or heuristic baselines. A typical result is a 50% relative reduction of classification error, from the majority baseline [22, 44, 53]. Some authors also report ROC curves [85, 112], or performance at different false acceptance rates [44, 52, 102], since the costs for false acceptances and false rejections are different, and most likely domain-specific.

5.3 Problem statement

Let R be a user input, typically characterized in terms of a set of features from different knowledge sources in the system $R=<f_1, f_2, \ldots, f_n>$. Let C be a binary variable which encodes whether or not the system correctly understood the user (C=1 represents correct understanding, C=0 represents incorrect understanding). The semantic confidence annotation problem can then be formulated as follows:

> given a user input $R=<f_1, f_2, \ldots, f_n>$, compute the probability that the system correctly understood the user $P(C=1|R)$.

A restricted binary version of this problem can also be formulated:

> given a user input $R=<f_1, f_2, \ldots, f_n>$, predict whether or not this input was correctly understood by the system.

This second formulation of the problem, also known as misunderstanding detection, is a binary classification task. A significant body of research has been dedicated to this latter problem [18, 38, 53, 70]. However, in the context of conversational spoken language interfaces, a continuous con-

fidence score that accurately reflects the probability of correct understanding can provide a much more informative basis for error handling than a simple binary indicator. The probability of correct understanding can be used (in conjunction with costs for various system actions) in a decision theoretic framework for engaging in various error recovery strategies. We therefore argue that, in the context of spoken language interfaces, the focus should be on the first rather than the second formulation of the confidence annotation problem. Ideally, we would like to generate continuous confidence scores that accurately reflect the probability of correct understanding. In the following subsection we discuss two desirable properties for confidence scores – calibration and refinement, together with a corresponding evaluation criterion.

5.3.1 Evaluation criteria

5.3.1.1 Classification error and accuracy

Perhaps influenced by the binary formulation of the confidence annotation problem, the quality of confidence annotators is often evaluated using metrics such as **average classification error** (Avg.CE), its complement – **average classification accuracy** (Avg.CA) [18, 38, 53, 70]. If the confidence annotator outputs a binary label indicating correct or incorrect understanding B_i, these metrics are defined as follows:

$$\text{Avg.CA} = \sum_i C_i \cdot B_i + (1 - C_i) \cdot (1 - B_i)$$

$$\text{Avg.CE} = \sum_i C_i \cdot (1 - B_i) + (1 - C_i) \cdot B_i$$

Sometimes a combined measure of precision and recall (e.g. an F-measure), or ROC curves are reported for the constructed classifiers [44, 52, 85, 102, 112]. The use of these metrics is common even when the confidence annotator produces a continuous confidence score y_i . A threshold (th) is applied to derive a binary classification; by varying this threshold, different trade-offs between false-acceptances and false-rejections can be achieved. Average classification accuracy and/or average classification error are then computed as follows:

$$\text{Avg.CA} = \sum_i C_i \cdot (y_i > \text{th}) + (1 - C_i) \cdot (y_i \le \text{th})$$

$$\text{Avg.CE} = \sum_i C_i \cdot (y_i \le \text{th}) + (1 - C_i) \cdot (y_i > \text{th})$$

These metrics operate on a linear scale and the results are easy to interpret. For instance, an improvement from 60% to 80% in accuracy can be considered sizable. A t-test can be used to detect whether the performance differences between two annotators are statistically significant.

5.3.1.2 Calibration, refinement and the Brier score

As we have argued at the beginning of this section, spoken dialog systems make important error handling decisions based on confidence scores. In this context, a continuous score that accurately reflects the probability of correct understanding is more desirable than a binary estimate of correct understanding. The ideal confidence annotator acts as a probability forecaster. Two properties are desirable in this context: **calibration** and **refinement** [28, 32].

A confidence annotator (or a probability forecaster in general) is said to be **calibrated** if, as the number of predictions goes to infinity, the predicted probability $P(C=1|R)$ corresponds to the empirical probability of correct understanding. In other words, in the limit, the system correctly understands the user in y% of the cases when the confidence annotator assigns a score of $P(C=1|R)=y$. Calibration therefore measures to which extent the predicted probability accurately reflects the empirical probability of correct understanding.

The second important property is **refinement**. A confidence annotator that always predicts the average rate of correct understanding is perfectly calibrated, but also completely useless. Refinement measures the usefulness of each forecast. The more concentrated $P(C=1|R)$ is towards 0 or 1,

the more refined the confidence annotator [28]. A well-refined and well-calibrated confidence annotator always predicts probabilities close to 0 or 1, and these probabilities correspond to the empirical probabilities of correct understanding.

Metrics that capture both calibration and refinement are called **proper scoring rules**. Two such proper scoring rules are the **log-loss** (LL) [40] and the **Brier score** (Br) [28]. They are computed as follows:

$$\text{Avg.LL} = \sum_i C_i \cdot \log(y_i) + (1 - C_i) \cdot \log(1 - y_i)$$

$$\text{Avg.Br} = \sum_i C_i \cdot (1 - y_i)^2 + (1 - C_i) \cdot y_i^2$$

In the limit both these metrics accurately reflect the quality of the confidence annotation predictions. However, when only a small number of data points is available for the evaluation, the log-loss scoring function is less robust. A single prediction error can introduce a fairly large loss (because of the log function), and can significantly lower the overall, average log-loss. For this reason, we will use the Brier score in the rest of this work for assessing the confidence annotation predictions. To facilitate comparison with other works, and to provide a fuller understanding of the performance of the developed confidence annotators, we also report classification error. In section 5.4.2.5, we discuss the relationship between these metrics, based on empirical results in three different dialog domains.

Note that each of the metrics discussed above penalizes the confidence annotator's predictions with a certain loss function (cost structure). The loss functions are illustrated in Figure 58. The classification error metric uses a threshold of 0.5 and penalizes errors equally according to a zero-one loss function. The log-loss metric penalizes the predictions with a log-loss function. The Brier score corresponds to using a quadratic-loss function. The actual dialog-level loss function (cost structure)

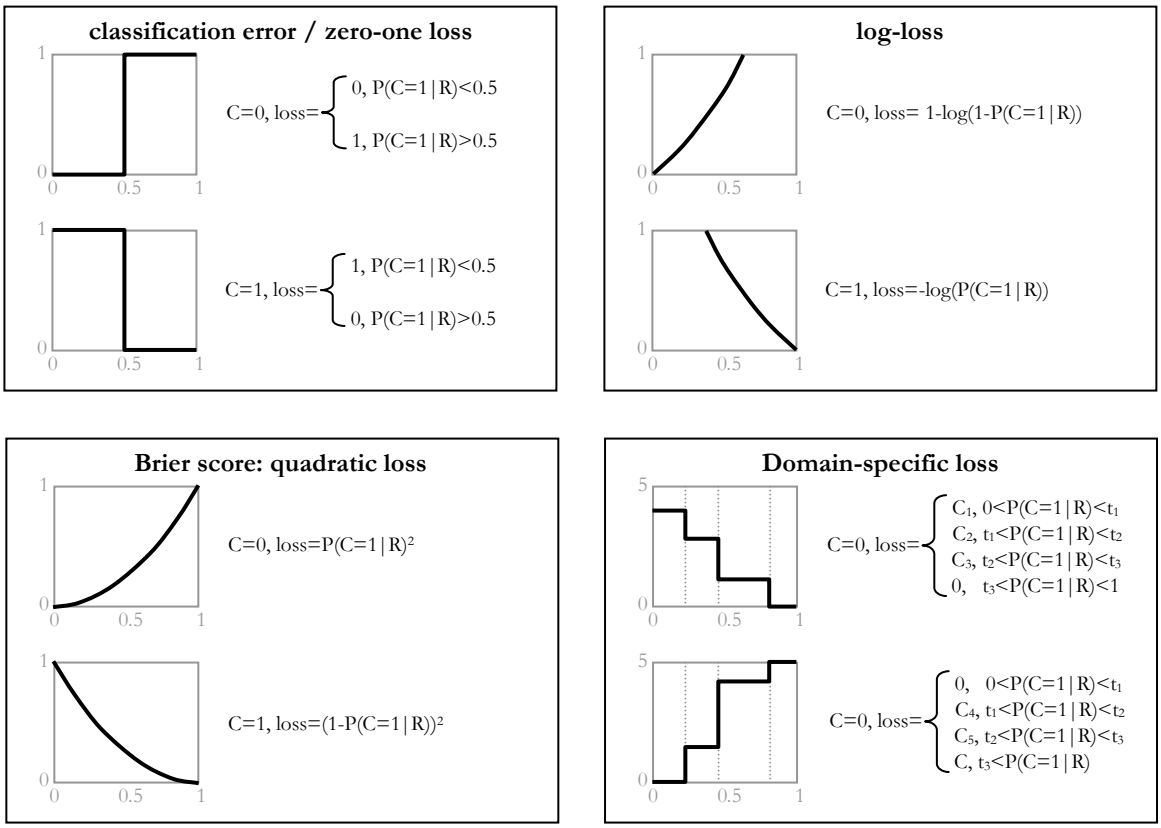


Figure 58. Loss functions for confidence annotation evaluation metrics

depends on the set of actions that dialog controller might engage, and on the relative costs of these actions. For instance, a commonly encountered approach is to use a set of thresholds t_1, t_2, t_3 on the confidence score and take different confirmation actions depending on how the confidence score $P(C=1|R)$ compares to these thresholds. If $P(C=1|R)<t_1$ the system rejects the input. If $t_1<P(C=1|R)<t_2$, the system engages in an explicit confirmation. If $t_2<P(C=1|R)<t_3$, the system engages in an implicit confirmation. Finally, if $t_3<P(C=1|R)$, the system accepts the input without any further confirmation. In this case, the actual cost structure is shown in the bottom right image from Figure 58. The costs are most likely asymmetric (e.g. the cost of rejecting a correct input is different from the cost of accepting an incorrect input). It is important to notice that, if we can generate well-calibrated confidence scores, we don't need to consider these domain-specific costs while optimizing or evaluating the confidence annotator. The annotator will produce scores that accurately reflect the probability of correct understanding, and these scores can be used at a later stage in conjunction with the domain-specific costs for making dialog control decisions.

5.4 A supervised learning approach for confidence annotation

In the previous section, we have formally introduced the problem of semantic confidence annotation and we have discussed several metrics for evaluating performance on this task. In this section, we discuss supervised learning approaches for building confidence annotation models. We begin by outlining the general methodology in subsection 5.4.1. Then, in subsection 5.4.2 we empirically investigate the use of four supervised learning techniques for building confidence annotation models using data from three different dialog domains. We address issues such as performance, sample-efficiency and cross-domain generalization of the proposed models. In subsection 5.4.3 we draw a number of conclusions based on these experiments.

5.4.1 Method

The semantic confidence annotation problem can be cast as a probabilistic prediction task:

> given a user input $R=<f_1, f_2, \ldots, f_n>$, compute the probability that the system correctly understood the user $P(C=1|R)$.

The supervised learning methodology for building a confidence annotation models is:

(1) collect a training corpus and label each utterance as either misunderstood or correctly-understood by the system;

(2) extract informative features from different knowledge sources in the system;

(3) train the confidence annotator on the collected corpus;

(4) evaluate the confidence annotator.

In the next section we describe a series of experiments in which we used this methodology to construct semantic confidence annotators in three different dialog domains.

5.4.2 Experimental results in the RoomLine, Let's Go!, and Let's Go! Public domains

5.4.2.1 Systems

We conducted experiments for building semantic confidence annotators based on three corpora from different spoken dialog systems: RoomLine, Let's Go!, and Let's Go! Public. RoomLine is a telephone-based, mixed-initiative spoken dialog system that can assist users in making conference room reservations on the CMU campus. The Let's Go! and the Let's Go! Public systems provide bus route and schedule information in the greater Pittsburgh area (the Let's Go system was an earlier

prototype of the Let's Go! Public system.) More information about the functionality of these systems was presented earlier, in subsections 3.4.1 and 3.4.2 from Chapter 3.

The RoomLine, Let's Go!, and Let's Go! Public systems were constructed using the same RavenClaw/Olympus infrastructure and components described in section 3.3 from Chapter 3 and in [12]. This input processing pipeline for these systems is illustrated in Figure 59. For speech recognition, the systems used 2 parallel Sphinx-II decoders [58]. Both decoders used the same trigram language model, but separate gender-specific acoustic models. The Sphinx-II recognition engines already provide a primitive form of confidence annotation: words in the recognition result where the language model was forced to back-off from a trigram to a bigram were marked as unconfident (see words surrounded by '?' in Figure 59). After decoding, the top-level hypothesis from each recognizer was forwarded to the language-understanding module. The Phoenix [131] robust semantic parser was used to generate a semantic representation for each of these hypotheses. The parse results for each hypothesis were then sent to Helios, the component responsible for confidence annotation. Helios assigned a confidence score to each hypothesis and forwarded the hypothesis with the highest confidence score to the dialog manager.

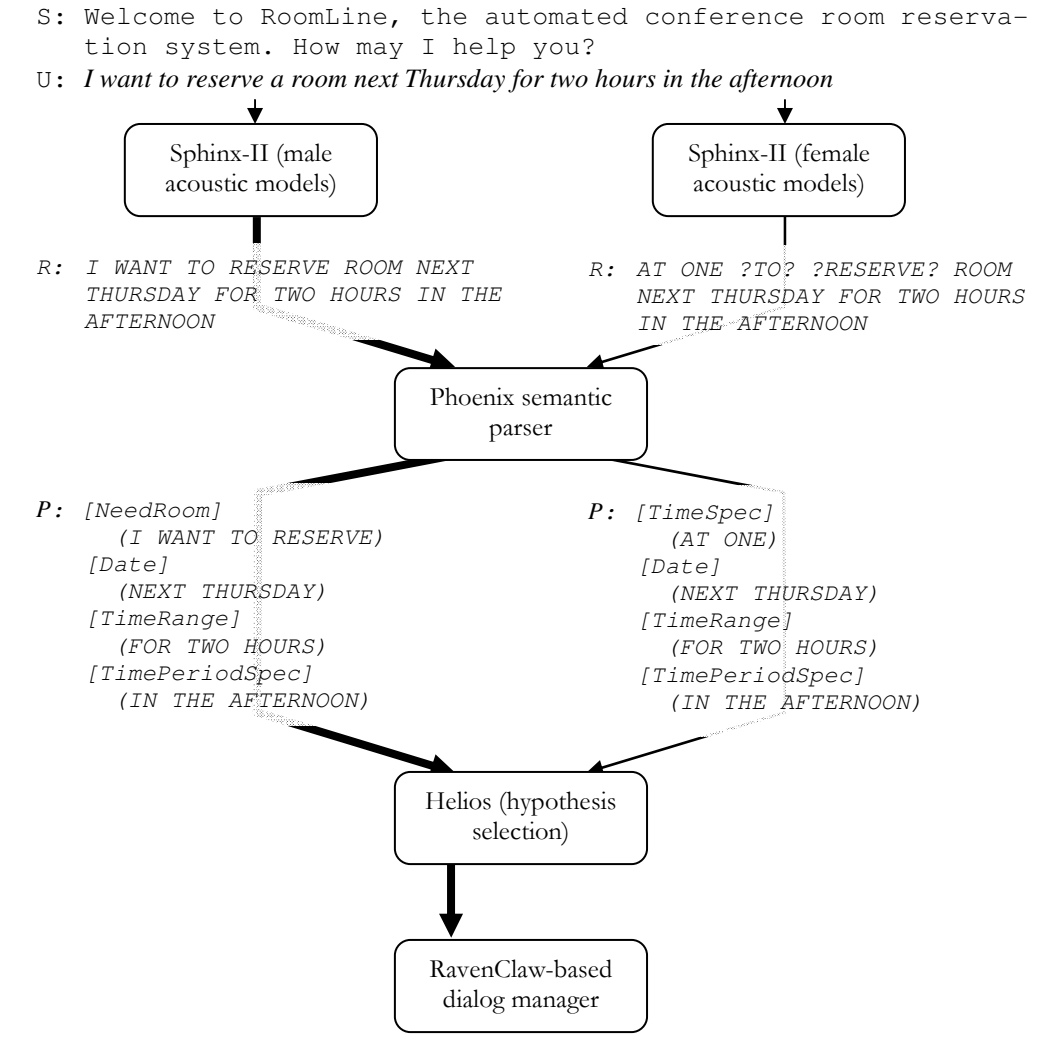


Figure 59. Input processing in the RoomLine, Let's Go!, and Let's Go! Public systems (`S:` marks the system turn; `U:` marks the user response; `R:` marks the recognition result; words tagged with '?' in the recognition result were marked as unconfident by the recognizer; `P:` marks the semantic representation of the recognition result)

5.4.2.2 Data

The RoomLine corpus was collected in a controlled user study described in more detail in section 8.3 from Chapter 8. During this study, 46 first-time users performed up to 10 scenario-driven interactions with the system. The resulting corpus contains 484 dialog sessions. All the non-understandings from this corpus were eliminated when training the confidence annotation models because confidence scores are not required for the non-understood utterances. The remaining corpus contains 8037 utterances. Of these, 1523, or 18.95% are misunderstandings. To facilitate comparison between the corpora, these numbers are also shown in Table 14.

The second corpus was collected with an early prototype version of the Let's Go! Bus Information system, before this system became available to the larger public. The system was advertised to the CMU community, and as a result this corpus contains calls made by various students and faculty on campus, as well as system developers. The total number of dialog sessions in this corpus is 874. After eliminating the non-understandings, the remaining corpus contains 4667 user turns, out of which 1489, or 31.90% are misunderstandings.

Finally, the third corpus was collected with the public version of the Let's Go! Bus Information system. In March 2005 this system was connected to the Pittsburgh Port Authority customer service line during non-business hours, and therefore became available to the larger Pittsburgh community. We used a corpus of 617 dialog sessions collected during the first month of operation for the system. After eliminating the non-understandings, the remaining corpus used for training contains 6029 utterances, out of which 1863, or 30.90% are misunderstandings.

	RoomLine	Let's Go!	Let's Go! Public
Dialog system domain	conference room reservations	bus schedule and route information	bus schedule and route information
Initiative type	mixed	mixed	system
Number of sessions	484	847	617
Number of utterances	8037	4667	6029
Number of misunderstandings	1523	1489	1863
Percentage misunderstandings	18.95%	31.90%	30.90%

Table 14. Basic statistics for confidence annotation corpora in the RoomLine, Let's Go! and Let's Go! Public systems

5.4.2.3 Features

We considered a large set of features extracted from different knowledge sources in the system. Below, we give a brief overview of the feature set. The full set of features is presented in Table 15.

- **speech recognition features.** We considered features such as the acoustic and language model scores, the number of words and frames in the recognized hypothesis, as well as word-level confidence information generated by the Sphinx recognizer (Sphinx tags each word in the decoded results as confident/unconfident, depending on the language model back-off pattern). We also considered a number of additional acoustic features such as the signal and noise-levels (as reported by the recognizer), and the speech-rate (computed on a per-word and per-syllable basis).
- **prosody features.** We included a large set of prosody features such as various pitch characteristics (mean, max, min, standard deviation, min and max slopes, etc.). These features were extracted in batch mode using the `get_f0` program.
- **lexical features.** We identified the top-10 words most correlated with misunderstandings, and added binary features to capture the presence or absence of these words (note that these lexical features are system-specific).

Table 15. Features for confidence annotation

A superscript on the feature name indicates that the feature is specific to a certain dataset.
The feature types are encoded as follows: R=real, C=count, N=nominal, B=boolean
The derived features are encoded as follows:
norm = normalized version of the feature (z-score)
>m = binary version indicating if the feature value is greater than the mean value of the feature in the dataset
>0 = binary version indicating if the feature value is greater than 0
>1 = binary version indicating if the feature value is greater than 1
>2 = binary version indicating if the feature value is greater than 2
>4 = binary version indicating if the feature value is greater than 4
dtf = difference between the current feature value and the feature value in the first turn in the dialog
dtp = difference between the current feature value and the feature value in the previous turn in the dialog

Feature name	Type	Derived features	Feature Description
Speech recognition features			
engine_id	C		which recognition engine (male or female) generated this hypothesis
am_score	R	norm, >m	the acoustic model score
lm_score	R	norm, >m	the language model score
decoder_score	R		the decoder score
acoustic_gap^{LG,LGP}	R	>m	Indicates the difference between the current acoustic model score and the acoustic score corresponding to an all-phone model
min_word_conf^{LG,LGP}	R		the minimum word-level confidence score
avg_word_conf^{LG,LGP}	R		the average word-level confidence score
max_word_conf^{LG,LGP}	R		the maximum word-level confidence score
frame_num	C	>m	the number of frames
word_num	C	>1, >2, >4	the number of words in the utterance
word_num_class	N		nominal feature indicating whether the utterance Rains 1 word, 2 words, 3 or 4 words, or more than 4 words
unconf_num	C	norm, >0	number of unconfident words (Sphinx tags individual words as unconfident if no trigram is found in the language model ending in the current word, and a bigram back-off is forced)
unconf_ratio	R		the percentage of unconfident words in the hypothesis
conf_num	C	norm	number of confident words
speak_rate	R	>m	speech rate, computed as number of frames per word
speak_rate_phones	R	>m	speech rate, computed as number of frames per phone
speak_rate_syl	R	>m	speech rate, computed as number of frames per syllable
npow	R	>m	noise level
pow	R	>m	signal level
pow_npow_diff	R	>m	Signal-to-noise ratio
Prosody features			
pitch_mean	R	>m, dtf, dtp	the pitch mean
pitch_range	R	>m, dtf, dtp	the range of the pitch (max – min) for the utterance

pitch_min	R	>m, dtf, dtp	the minimum pitch level in the utterance
pitch_max	R	>m, dtf, dtp	the maximum pitch level in the utterance
pitch_std	R	>m, dtf, dtp	the pitch standard deviation
pitch_min_slope	R		the minimum value for the pitch slope in this utterance (the largest pitch decrease)
pitch_max_slope	R		the maximum value for the pitch slope in this utterance (the largest pitch increase)
num_voiced_segments	C		the number of voiced segments in the utterance
perc_unvoiced	R	>m	the percentage of the utterance (in frames) that is not voiced
prepau	R	>m	the length of the initial unvoiced segment (initial pause)
Lexical features			
mark_confirm	B		presence of confirmation markers
mark_disconfirm	B		presence of disconfirmation markers
mark_lex_bool	B		presence of confirmation or disconfirmation markers
$lex^{BL, LG, LGP}$	B		a set of binary features that captures the presence / absence of 10 words correlated with misunderstandings. The words are different for each corpus, and were selected via a corpus based computation of mutual information between the words and the misunderstanding labels
Language understanding features			
slot_num	C		number of grammar slots
rep_slots_num	C	>0	number of repeated grammar slots (wrt the previous turn)
new_slots_num	C	>0	number of new grammar slots (wrt the previous turn)
words_per_slot	R	>1, >2	average number of words per grammar slot
uncov_num	C	>0, norm	number of words not covered by the parse
uncov_conf_num	C	>0, norm	number of confident words not covered by the parse
uncov_ratio	R		the percentage of words not covered by the parse
uncov_conf_ratio	R		the percentage of unconfident words not covered by the parse
cov_num	C	norm	the number of words covered by the parse
frag_num	C	>1	the number of fragments in the parse
frag_ratio	R		the percentage of fragments in the parse
gap_num	C	>0, >1	the number of gaps in the parse
frag_and_gap_num	C	>1	the number of fragments and gaps in the parse
hyp_num_parses	C		the number of alternative parses generated for this recognition hypothesis (due to grammar ambiguities, Phoenix can sometimes generate multiple parses for a single recognition hypothesis)
total_num_parses	C		the total number of alternative parses generated for this user input
num_parses_ratio	R		hyp_num_parses divided by total_num_parses
Inter-hypotheses features			
ih_diff_lexical	B		the two recognition hypotheses from the male and female recognition engine are different
ih_am_score_diff_to_max	R	>0	the difference between the acoustic model score of the current hypotheses to the maximum acoustic model score of the two hypotheses
ih_am_score_diff_to_min	R	>0	the difference between the acoustic model score of the current hypotheses to the minimum acoustic model score of the two hypotheses
ih_lm_score_diff_to_max	R	>0	the difference between the language model score of the current hypotheses to the maximum language model score of

			the two hypotheses
ih_lm_score_diff_to_min	R	>0	the difference between the language model score of the current hypotheses to the minimum language model score of the two hypotheses
ih_frag_ratio_diff_to_max	R	>0	the difference between the fragmentation ratio of the current hypotheses to the maximum fragmentation ratio of the two hypotheses
ih_frag_ratio_diff_to_min	R	>0	the difference between the fragmentation ratio of the current hypotheses to the minimum fragmentation ratio of the two hypotheses
Dialog features			
slots_matched	C	>1	the number of grammar slots that matched an open dialog expectation
slots_blocked	C	>0	the number of grammar slots that matched a closed dialog expectation
first_level_matched	C	>0, >1	the first level in the expectation agenda where a slot from the current input matched an open expectation
last_level_matched	C	>0, >1	the last level in the expectation agenda where a slot from the current input matched an open expectation
last_level_touched	C	>0, >1	the last level in the expectation agenda where a slot from the current input matched a (open or closed) expectation
matched_in_focus	B		the input matched the dialog expectation in focus (the first level on the agenda)
barge_in	B		the user barge-in on the system
turn	C	>0, >1, >m	the turn number
dialog_state_4	N		indicates whether the current dialog state is an open request (e.g. How may I help you?), a request for a Boolean concept (e.g. Would you like this room?) or a request for a non-boolean concept (e.f. For what time would you like this room?)
dialog_state_5	N		Indicates whether the current dialog state is an open request (e.g. How may I help you?), a request for a specific concept (e.g. For what time would you like this room?) or an explicit confirmation (e.g. Did you say you wanted the room for 10 a.m.?)
dialog_state_id[BL, LG,LGP]	B		set of binary features capturing the state the dialog manager is in.
Dialog history features			
last_turn_nonu	B		indicates if the previous turn was a non-understanding
num_prev_nonu	C	>1	indicates how many consecutive non-understandings preceded the current user turn
num_prev_not_nonu	C	>1	indicates how many consecutive turns that were not non-understandings preceded the current turn
h_ratio_nonu	R	>m, wind	indicates the ratio of non-understandings up to this point in the dialog; the “wind” version indicates the percentage in the last 5 user turns (same holds true for all following features)
h_avg_uncov_num	R	>m, wind	indicates the average number of words uncovered by the parse up to this point in the dialog, and within the 5 last turns
h_avg_gap_num	R	>m, wind	indicates the average number of parse gaps up to this point in the dialog, and within the 5 last user turns
h_avg_slots_matched	R	>m, wind	indicates the average number of slots that match an open expectation up to this point in the dialog, and within the 5 last user turns
h_avg_am_score	R	>m, wind	indicates the average acoustic model score up to this point in the dialog, and within the 5 last user turns
h_avg_lm_score	R	>m, wind	indicates the average language model score up to this point in the dialog, and within the 5 last user turns

- **language understanding features.** We used various features describing the semantic parse constructed by the Phoenix parser: the number of slots, how many of these slots were new or repeated, as well as various measures of parse-fragmentation.
- **inter-hypotheses features.** Since two parallel decoders were used, we also computed a set of features describing the relationship between the top-most hypothesis from the two recognizers. For instance, we computed whether these two hypotheses were identical at the lexical level, as well as the difference between their acoustic-model, language-model and parse-fragmentation scores.
- **dialog management.** We used a number of features capturing the match between the recognized hypothesis and the dialog manager expectation, as well as system-specific features describing which state the dialog was in.
- **dialog history.** Finally, we also included a number of features capturing various aspects of the dialog history: the number of previous consecutive non-understandings, the ratio of non-understandings up to the current point in the dialog, and tallied averages of the acoustic-model, language-model, and various parse-fragmentation scores.

As Table 15 illustrates, various normalized versions of these features were also created (e.g. z-score normalization, binary normalization, etc.)

5.4.2.4 Supervised learning techniques

We conducted comparative experiments with four different supervised learning techniques:

- **Logistic regression models**. We used a stepwise approach for constructing logistic regression models [76]. In each step, the next most informative feature was added in the model, as long as the average data likelihood on the training set improved by a statistically significant (p-accept=0.05) margin. After each step, the p-values for all the features currently in the model were reassessed, and any feature with a p-value larger than p-reject=0.3 was eliminated from the model. To avoid over-fitting to the training data, we used Bayesian Information Criterion as a stopping criterion.
- **CART (Classification and Regression Trees)**. We constructed regression trees [74] using Gini's diversity index [134] as a splitting criterion. Tree pruning was performed based on a 10-fold cross-validation procedure.
- **AdaBoost**. We used Shapire's Adaboost.M1 algorithm [37] with simple decision stubs as weak learners, and 100 boosting stages.
- **Naïve Bayes**. We constructed a Naïve Bayes classifier [74] using the full set of features described in the previous section.

We have previously argued that in the context of spoken dialog systems we are interested in generating continuous confidence scores that accurately reflect the probability of correct understanding. In other words, we need to generate well-calibrated class posterior probabilities P(C=1 | R). Most discriminative classifiers do not automatically generate well-calibrated outputs. The problem can be addressed by recalibrating the classifier outputs. A relatively simple method[15] for post-calibration is to fit a sigmoid function that transforms the classification output into a probability score. If the classification model M generates output M(R) for input R, then we can fit a sigmoid such that:

$$P(C=1 \mid R) = \frac{e^{\alpha+\beta \cdot M(R)}}{1+e^{\alpha+\beta \cdot M(R)}} \qquad \textbf{[1]}$$

This post-calibration step is not necessary for logistic regression, since the logistic regression model has this calibration step embedded. In fact, logistic regression directly models P(C=1 | R) as in equation [1], using a weighted linear feature combination to describe M(R). Post-training calibration was therefore performed for the other three supervised learning techniques: the regression tree, the

[15] More complex methods for calibration have been recently proposed and evaluated [7].

AdaBoost model and the Naïve Bayes model. For these models, we report below both calibrated and un-calibrated results.

5.4.2.5 Experimental results

§ Individual feature analysis

We began by investigating the informative power of individual features. Each feature was used to construct a simple logistic regression model for predicting whether the utterance was misunderstood or not. The features were then ranked according to the average Brier score in a 10-fold cross-validation process. The top-25 most informative features for each dataset are shown in Table 16.

This preliminary analysis confirms that informative features can be extracted from different knowledge sources in the system. A large number of features from the speech recognition, language understanding, and dialog management levels appear in the top-25 list. Additionally, a few inter-hypothesis, prosody, lexical, and dialog history features are also present in the top-25. Some of the most informative features come from the dialog management level, and they describe how well the decoded hypothesis corresponds to the dialog manager expectation (e.g. `first_level_matched`, `last_level_matched`, `last_level_touched`, `expectation_match`, etc.). Language understanding features, such as goodness-of-parse scores (e.g. `uncov_num_bool`, `gap_num`, etc.) also carry relevant information. Although the majority of these features are informative for more than one domain, the identity and relative ranking of the top-25 features is not the same across the three domains. For instance, in the RoomLine domain the top-10 list contains dialog management, lexical, inter-hypothesis, language understanding and speech recognition features. In the Let's Go! domain, the top-10 list is dominated by speech recognition features. Finally, in the Let's Go! Public domain, the dialog management features seem to be the most informative. In conclusion, although similarities exist, no small domain-independent subset of informative features can be easily identified. In the rest

RoomLine		Let's Go!		Let's Go Public!	
DM	last_level_matched	IH	ih_diff_lexical	SR	acoustic_gap
LEX	mark_lex_bool	SR	min_validword_conf	DM	last_level_matched>0
IH	ih_diff_lexical	SR	avg_validword_conf	DM	last_level_touched>0
DM	first_level_matched	SR	avg_word_conf	DM	first_level_matched>0
LU	uncov_num_bool	SR	min_word_conf	DM	expectation_match
LU	gap_num	SR	am_score_norm	DM	matched_in_focus
LU	frag_ratio	SR	lm_score	DM	last_level_touched
LU	uncov_ratio	SR	decoder_score	SR	am_score_norm
LU	uncov_num_norm	SR	am_score	PR	num_voiced_segments
SR	unconf_num_bool	LU	uncov_num_bool	DM	last_level_matched
SR	unconf_ratio	LU	gap_num	SR	lm_score
LU	frag_and_gap_num	LU	uncov_num_norm	DM	first_level_matched
DM	last_level_touched	LU	frag_and_gap_num	SR	decoder_score
DM	last_level_matched>0	LU	frag_ratio	SR	am_score
DM	last_level_matched>1	SR	unconf_num_norm	SR	acoustic_gap>m
SR	unconf_num_norm	DH	h_avg_am_score_norm	SR	am_score_norm>m
DM	last_level_touched>0	PR	num_voiced_segments	SR	frame_num
DM	last_level_touched>1	SR	frame_num	LEX	lex(YES)
LEX	lex(YES)	LU	uncov_ratio	IH	ih_am_score_norm_diff_to_min
DM	expectation_match	SR	unconf_num_bool	LEX	mark_confirm_bool
DM	first_level_matched>0	SR	unconf_ratio	DM	last_level_touched>1
SR	lm_score	SR	unconf_num	LEX	mark_lex_bool
DM	first_level_matched>1	DH	h_wind_avg_am_score_norm	IH	ih_am_score_norm_diff_to_min_bool
SR	frame_num_gtm	SR	acoustic_gap_gtm	SR	word_num_norm
LEX	mark_confirm_bool	SR	am_score_norm_gtm	SR	word_num

Table 16. Top 25 most informative features for semantic confidence annotation in the RoomLine, Let's Go! and Let's Go! Public domains (DM – dialog management features; SR – speech recognition features; LU – language understanding features; IH – inter-hypothesis features; PR – prosody features; DH – dialog history features; lightly shaded cells contain features present in the top-25 list for 2 of the 3 domains; the darkest shaded cells contain features present in the top-25 list for all 3 domains)

of this work we therefore consider the full list of features (presented in Table 15) for building the confidence annotation models.

§ Results

Next, we trained confidence annotators for each of the 3 corpora, using the four machine learning techniques outlined in the previous subsection. For each dataset and machine learning technique, we built two models: one that used the full set of features available for that dataset (FULL), and one that used only the features available across all datasets (COMM). The models were evaluated by computing classification-error and the Brier score in a 20-fold cross-validation procedure (20 random permutations of each dataset were generated, and 500 points were held-out for testing from each permutation.) The results are presented in Table 17 and illustrated in Figure 60; the figure shows only results on the COMM models.

Several consistent patterns can be observed across domains. First, the models using the full set of features (FULL) generally perform better than the models using only the common set of features (COMM) – see Table 17. This is an expected result; we have already seen that some domain-specific features carry relevant information for the confidence annotation task. The performance gap is however not very large, indicating that successful models can be built using only domain-independent features.

The logistic regression model performs best overall. When evaluated in terms of classification error, the decision tree and the AdaBoost models perform similarly well. The Naïve Bayes model performs significantly worse than the other three models in all cases. In the RoomLine domain, the Naïve Bayes model performs worse than the majority baseline. This result is in line with our observations in an earlier confidence annotation experiment on the CMU Communicator data [22]. We believe the explanation lies in part in the fact that the independence assumption made by the Naïve Bayes model is violated by our feature set.

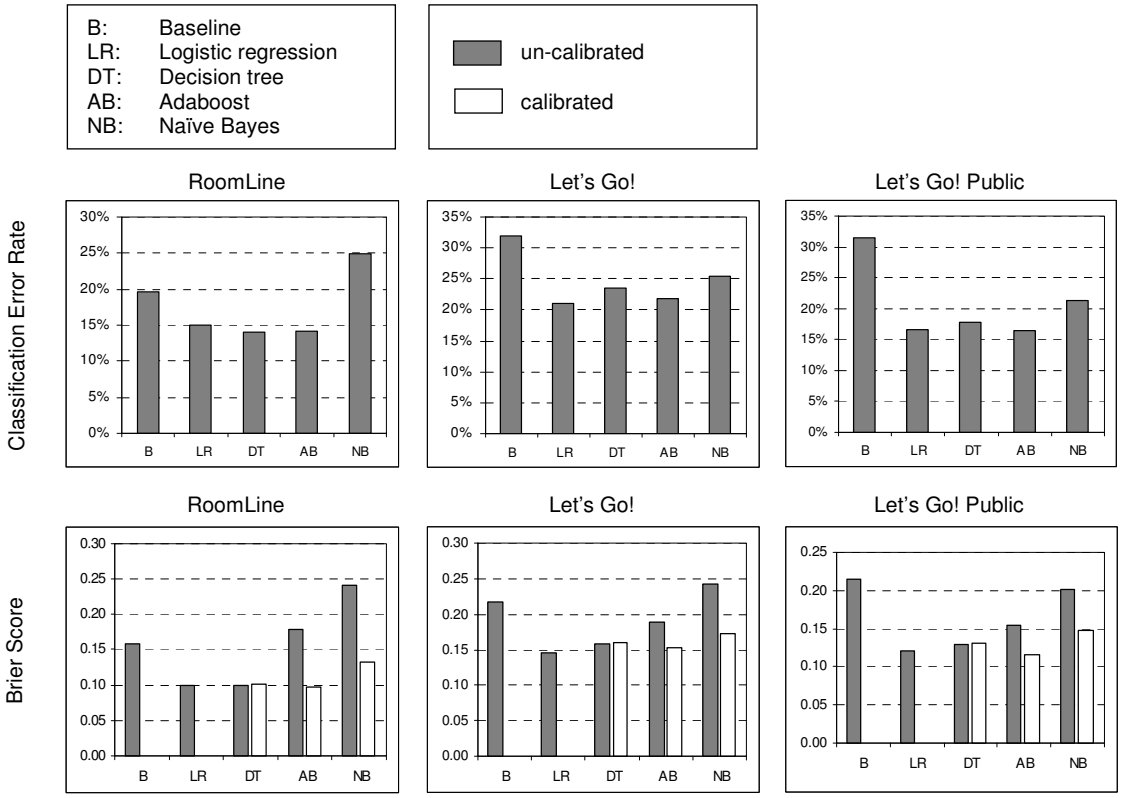


Figure 60. Confidence annotation model performance in the RoomLine, Let's Go!, and Let's Go! Public domains (classification error and Brier score for calibrated and un-calibrated classifiers)

A. Classification error

Corpus	RoomLine		Let's Go!		Let's Go! Public	
Majority baseline	19.6%		32.0%		31.4%	
Feature set	COMM	FULL	COMM	FULL	COMM	FULL
Logistic regression	14.9%	12.8%	21.1%	20.1%	16.7%	16.7%
Decision Tree	13.9%	13.6%	23.6%	21.6%	17.7%	17.3%
Adaboost	14.1%	12.6%	21.8%	20.3%	16.5%	16.2%
Naïve Bayes	25.0%	24.8%	25.4%	24.2%	21.4%	20.7%

B. Brier score

Corpus	RoomLine		Let's Go!		Let's Go! Public	
Majority baseline	0.1574		0.2175		0.2156	
Feature set	COMM	FULL	COMM	FULL	COMM	FULL
Logistic regression	0.0989	0.0873	0.1454	0.1373	0.1200	0.1171
Decision Tree	0.0985	0.0969	0.1581	0.1529	0.1290	0.1276
Dec. Tree (calibr.)	0.1020	0.1012	0.1604	0.1549	0.1314	0.1300
Adaboost	0.1783	0.1422	0.1884	0.1778	0.1542	0.1536
Adaboost (calibr.)	0.0976	0.0896	0.1518	0.1419	0.1159	0.1151
Naïve Bayes	0.2420	0.2391	0.2426	0.2319	0.2011	0.1954
N. Bayes (calibr.)	0.1325	0.1306	0.1731	0.1682	0.1479	0.1445

Table 17. Confidence annotation performance

Although the classification error results indicate that the logistic regression, decision tree and AdaBoost models perform similarly well, the Brier score evaluation reveals a different state of affairs. The AdaBoost model has a significantly worse Brier score than the logistic regression model and the regression tree. However, the performance of the post-calibrated AdaBoost model equals the performance of the logistic regression model. Figure 61 provides a more detailed picture of the Brier score evolution of the AdaBoost model. As the number of boosting stages increases, the classifier focuses on the harder examples and strives to maximize their margin. In this process, the classifier becomes less and less calibrated (after an initial decrease, the Brier scores increases with the number of boosting stages). Recalibrating the resulting classifier corrects this problem. For the Naïve Bayes classifier, the post-calibration procedure produces similar significant improvements. No improvement was attained for the regression tree.

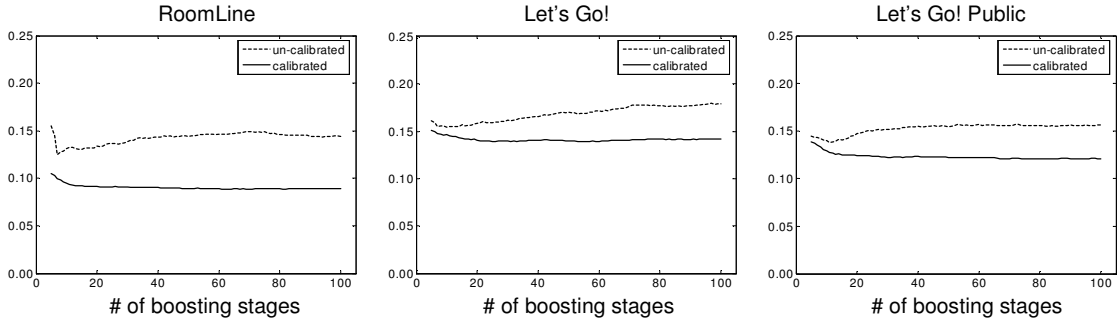


Figure 61. Brier score evolution as a function of the number of boosting stages (Adaboost model)

§ Model analysis

Next, we inspected the resulting logistic regression models in order to better understand which features were most informative when considered in combination. Table 18 shows the model constructed from the RoomLine corpus. The coefficients describe the effect of each feature on the log-odds of misunderstanding. A positive coefficient indicates a feature that increases the likelihood of misunderstanding; a negative coefficient indicates a feature that decreases that likelihood. The last column in the table shows the sign of the coefficient.

As expected, several of the features that performed very well individually are present in this

RoomLine			
	Feature	Coef.	Effect
	k	-17.58	-
LEX	mark_lex_bool	-2.55	-
IH	ih_diff_lexical	1.45	+
DM	last_level_matched	0.54	+
SR	am_score_norm	-0.00	-
DM	pragmatic_flag	2.01	+
LU	uncov_ratio	3.51	+
SR	lm_score_norm_2	0.00	+
DM	dialog_state_id:RequestStartTime	1.46	+
LU	slot_num	0.74	+
DM	dialog_state_id:RefineSize	-1.37	-
DM	dialog_state_id:RequestEndTime	0.90	+
IH	ih_am_score_norm_diff_to_min	-0.00	-
LU	uncov_num	-0.28	-
DH	h_avg_gap_num>m	0.44	+
LU	slots_relevant_num>1:	0.86	+
SR	lm_score_norm_1:	-0.00	-
SR	speak_rate_syl:	-44.37	-
DM	dialog_state_id:RequestDate	1.08	+

Table 18. Confidence annotation logistic regression model in the RoomLine domain (DM – dialog management features; SR – speech recognition features; LU – language understanding features; IH – inter-hypothesis features; DH – dialog history features)

model: `mark_lex_bool`, `ih_diff_lexical`, `last_level_matched`, etc. At the same time, a number of additional features become informative when considered in conjunction with other features. For instance, the model contains 4 dialog-state features that were not present in the top-25 list from Table 16. According to the learned model, misunderstandings are more likely to happen when the system asks for the `start_time`, the `end_time` and the `date` concepts, and less likely to happen when the system asks the user whether they would like a small or a large room (`dialog_state: RefineSize`). A decoded result is more likely to be a misunderstanding if there is a difference between the 2 parallel decoded hypotheses (`ih_diff_lex`), if the input matches an expectation that is placed in the lower levels of the expectation agenda (`last_level_matched`), if there is a domain-specific constraint violation (`pragmatic_flag`), if the number of slots in the decoded results is high (`slot_num`), and so on. The structure of the model corresponds therefore in a lot of ways to our intuitions about misunderstandings. At the same time, the precise weights are derived from data, and optimized for the task at hand.

§ Performance versus training set size

The three corpora we have used in the experiments described above ranged in size from 4500 to 8000 data-points. In other previous work reported in the literature the dataset sizes for training confidence annotation models ranged from hundreds [38, 52] to several tens of thousands of utterances [44, 85]. Labeled data is generally difficult and costly to acquire. As a consequence, it is interesting to better understand the relationship between training set size and performance. Would more data have helped build a better confidence annotator in our domains? Or is performance already reaching an asymptote? Which one of the supervised learning techniques discussed above is the most sample efficient?

To assess the relationship between training set size and performance, we trained successive models using increasingly larger amounts of data. The same 20-fold cross-validation process was used: first, we generated 20 random permutations of each dataset, and held-out 500 of these data-points for testing. Next, each of the 4 models discussed in the previous section were trained using increasingly larger amounts of training data: 100, 200, 300, 400, 500, 1000, 1500 samples, and so forth continuing in increments of 500 samples. In these experiments, the full set of features (`FULL`) was used in each domain. The evaluation was performed by computing average performance on the 500 held-out points, across the 20 permutations.

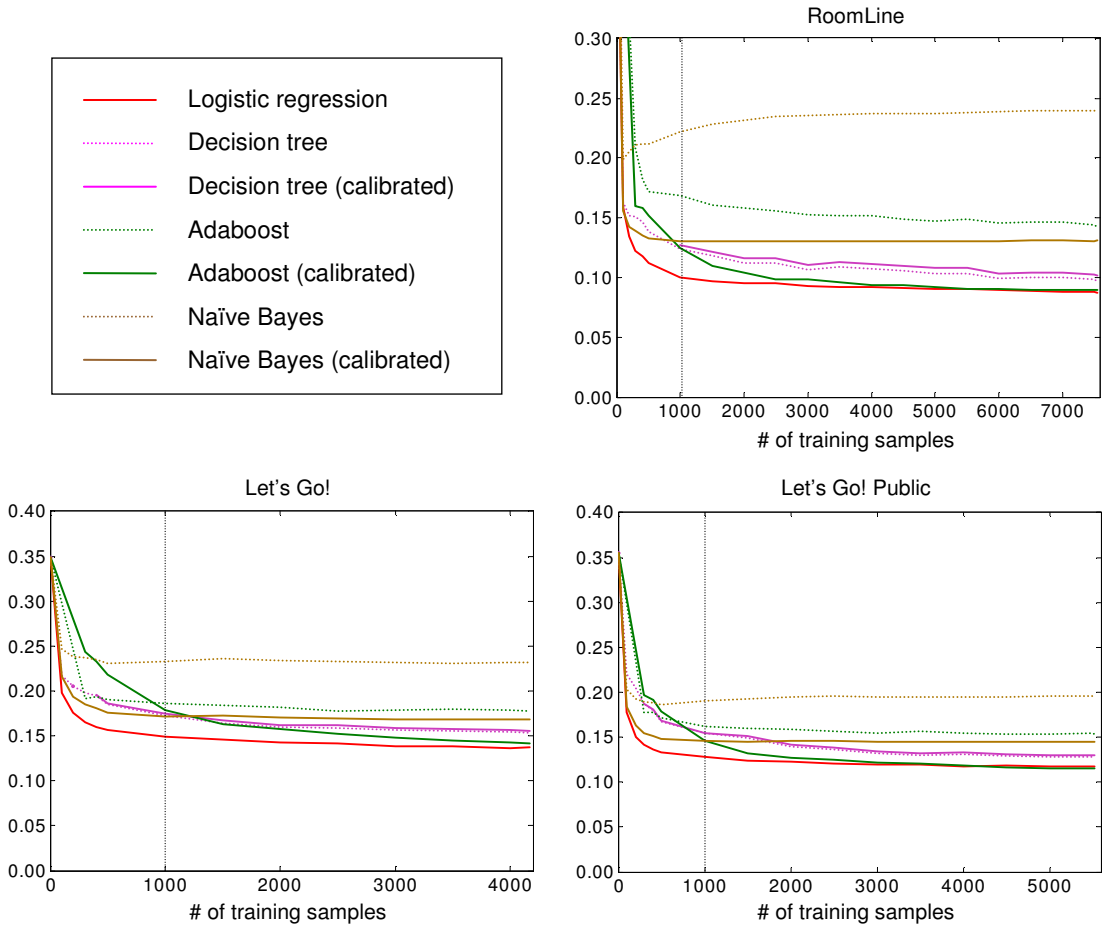


Figure 62. Confidence annotator performance (Brier score) as a function of training set size

The results are illustrated in Figure 62. In all three domains, the performance of the classifiers (measured as the Brier score) reaches an asymptote by the time we use the full training corpus. These results indicate that more training data will not further improve the quality of the confidence annotation model. In fact, for all models the largest part of the performance gain is obtained after using about 1 or 2 thousand training instances. Following that point, adding more data only marginally improves performance. The logistic regression model is the most sample efficient in all three domains. With only 500 training instances, this model significantly outperforms the other ones. Given its simplicity, good calibration, and good sample efficiency properties, from now on we will focus our attention only on the logistic regression confidence annotation models.

5.4.2.6 An investigation of cross-domain transferability

Supervised learning techniques require a pre-existing in-domain corpus of labeled data. For each new system developed, we need to collect a new corpus and train a new confidence annotation model. Unfortunately, collecting such corpora is generally costly and labor intensive. In this subsection, we investigate how well existing confidence annotation models generalize across new domains.

To investigate this question, we conducted a cross-domain evaluation of the learned confidence annotation models. We focused our attention on the COMM version of logistic regression models, since they use features available across all domains. We evaluated each model in the other two domains, and compared performance against the corresponding in-domain models.

The results are presented in Table 19, and illustrated in Figure 63. The models trained with data from the RoomLine and Let's Go! domains generalized relatively well to the other domains.

Corpus	RoomLine data		Let's Go! data		Let's Go! Public data	
	Brier sc.	%GAP	Brier sc.	%GAP	Brier sc.	%GAP
Baseline	0.1574	0%	0.2175	0%	0.2156	0%
RoomLine model	0.0989	100%	0.1668	70%	0.1610	57%
Let's Go! model	0.1146	73%	0.1454	100%	0.1521	66%
Let's Go! Public model	0.1400	30%	0.2031	20%	0.1200	100%

Table 19. Cross-domain evaluation of logistic regression confidence annotation models (in-domain model performance is in bold-face; %GAP indicates which percentage of the gap between the baseline Brier score and the in-domain model is covered by the cross-domain model)

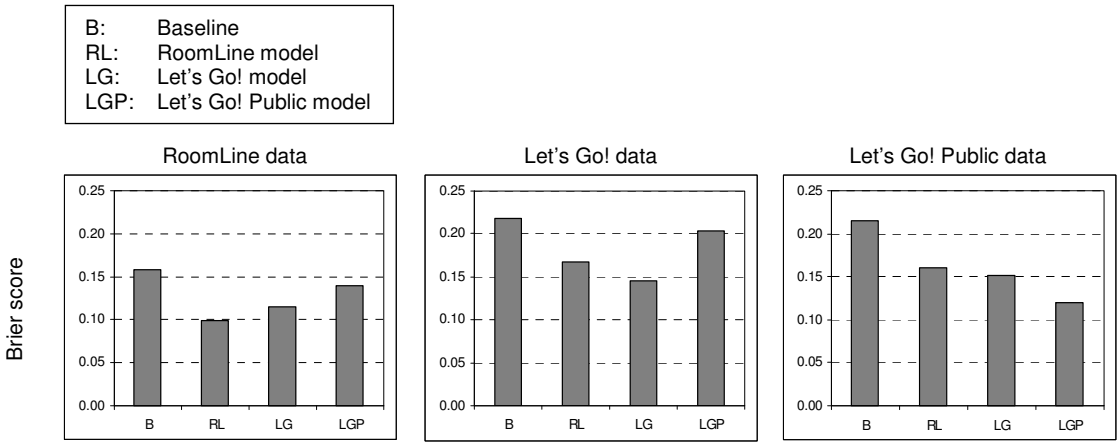


Figure 63. Cross-domain evaluation of confidence annotation models

They led to significant improvements over the majority baseline, covering 60-70% of the Brier score gap between the baseline model and the in-domain model – see Table 19. At the same time, the Let's Go! Public model did not generalize that well: it covered only 30% of the gap in the RoomLine domain and 20% of the gap in the Let's Go! domain. The improvements from the majority baseline are statistically significant in all cases with p-values smaller than 10^{-4}. These results indicate that confidence annotation models may sometimes generalize relatively well to other domains, but this is not always the case. Moreover, the transfer seems to be asymmetric: for instance, the RoomLine model generalizes relatively well to the Let's Go! Public domain, but the same is not true in reverse. A similar asymmetry can be observed between the Let's Go! and Let's Go! Public models.

We believe the explanation for the differences in how well the models generalize and the observed asymmetry lies in the nature of the supervised learning paradigm. Data-driven confidence annotation models are automatically tuned to the characteristics of the training set, and, unless the new domain has the same characteristics, the model will not perform as well as a model trained with in-domain data. For instance, a comparison of the word-error-rate (WER) distribution across the three corpora reveals significant differences between the Let's Go! Public corpus and the other two corpora – see Figure 64. There are a larger number of high WER utterances in the Let's Go! Public corpus. As a consequence, the Let's Go! Public model is focused more on these utterances (in comparison with the other two models.) When transferred into the other two domains, this model encounters less utterances with a high WER (on which it can presumably predict well) and more utterances with low WER (on which it falters more often). As a result the model does not generalize well. In contrast, when the RoomLine model is transferred into the Let's Go! Public domain, although the model does not perform as well on the high WER utterances, the majority of utterances is still low WER and overall the model transfers better. In Figure 65 we show the classification error rates of the RoomLine and Let's Go! Public models across both domains, for utterances in each of the three word-error-rate ranges introduced above: 0-40%, 40-80% and >80%. These plots corroborate our explanation.

Another potential explanation for the observed asymmetry is the automatic feature selection

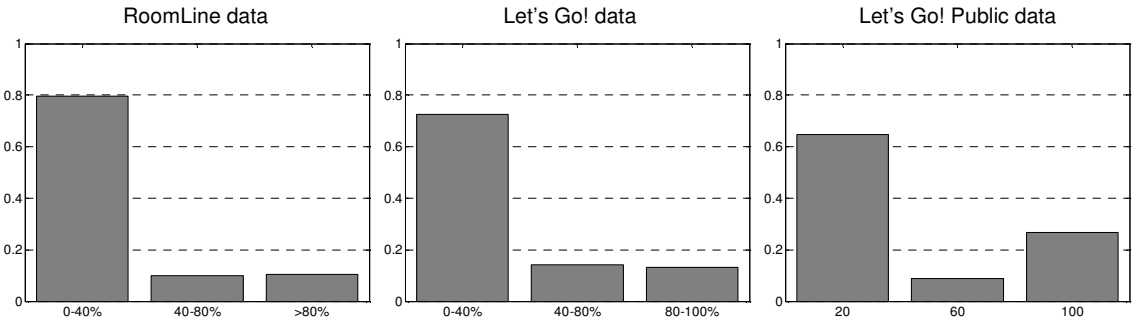


Figure 64. Different word-error-rate distribution in the RoomLine, Let's Go! and Let's Go! Public domains

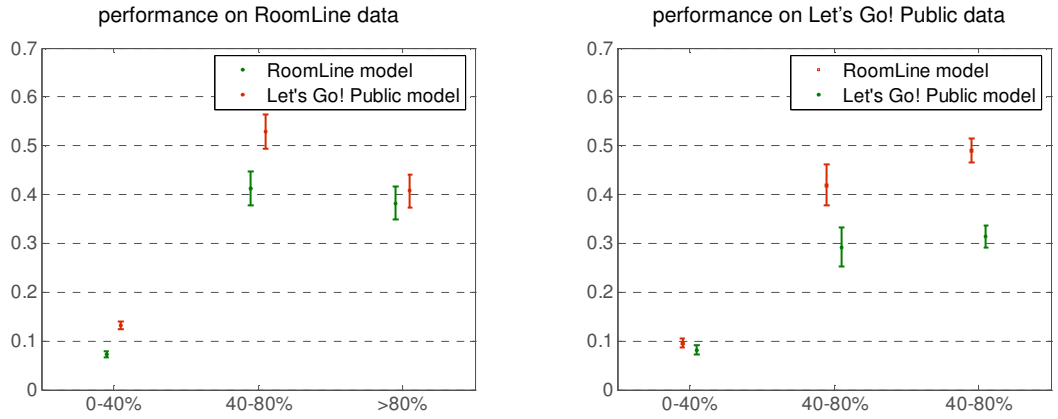


Figure 65. Cross-domain performance of RoomLine and Let's Go! Public models at different WER levels

process. Each model, when trained, selects a subset of the available features that is most informative for performing confidence annotation on the given training set. The features selected by the Let's Go! Public model are different than those selected by the RoomLine model, and the distributional similarity between the two datasets along those two sets of coordinates might also be significantly different (this could be one reason for the observed asymmetry.)

Since our results show that an automatic reliable transfer is not always possible, we focused our attention next on the following question: **how well can we do if we have access to a small amount of labeled data in the new domain?** (recall that our motivation is that labeling data is expensive, and that we would like to reuse or adapt an existing confidence annotator into a new domain with a minimal amount of developer effort.) We took a simple supervised adaptation based approach to this problem. We randomly considered 100 labeled data-points in the target domain, and recalibrated the confidence score produced by the out-of-domain model using this small amount of in-domain data. In other words, we build a calibration model that is trained to predict an adapted confidence score CA based on the original confidence score C produced by the model:

$$CA = \frac{e^{\alpha+\beta \cdot C}}{1+e^{\alpha+\beta \cdot C}}$$

The calibration model was trained using only 100 randomly selected labeled data-points from the target domain. Given the relatively small number of labeled data-points used (100), we will refer to the post-calibrated model as the `small`-calibration model. For comparison purposes, we also constructed a `full`-calibration model that uses all the data-points in the target domain to perform a recalibration. In addition, we also report the performance of an in-domain model trained using 100 data-points (i.e. what would happen if instead of adapting a model from a different domain by using 100 labeled data-points, we would directly train an in-domain confidence annotation model using

Corpus	RoomLine data		Let's Go! data		Let's Go! Public data	
	Brier sc.	%GAP	Brier sc.	%GAP	Brier sc.	%GAP
Majority baseline	0.1574	0%	0.2175	0%	0.2156	0%
100 data-points in-domain model	0.1545	5%	0.1911	37%	0.1687	49%
RoomLine model	0.0989	100%	0.1668	70%	0.1610	57%
RoomLine model + small calibration			0.1667	70%	0.1637	54%
RoomLine model + full calibration			0.1640	74%	0.1603	58%
Let's Go! model	0.1146	73%	0.1454	100%	0.1521	66%
Let's Go! model + small calibration	0.1211	62%			0.1445	74%
Let's Go! model + full calibration	0.1175	68%			0.1419	77%
Let's Go! Public model	0.1400	30%	0.2031	20%	0.1200	100%
Let's Go! Public model + small calibr.	0.1333	41%	0.1694	67%		
Let's Go! Public model + full calibration	0.1290	49%	0.1659	72%		

Table 20. Cross-domain evaluation of logistic regression confidence annotation models with calibration (in-domain model performance is in bold-face; %GAP indicates which percentage of the gap between the baseline Brier score and the in-domain model is covered by the cross-domain model)

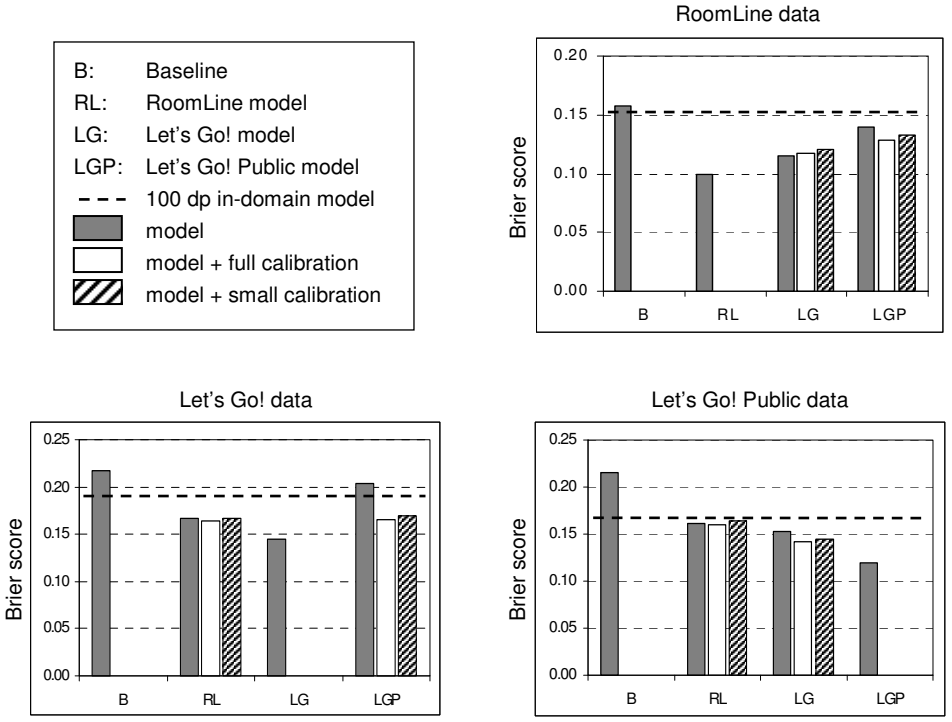


Figure 66. Cross-domain evaluation of confidence annotation models with calibration

those data-points?)

The results were computed using the same 20-fold cross-validation procedure and are shown in Table 20 and illustrated in Figure 66. Overall, the post-calibration procedure improves the performance of the confidence annotation models. In two cases, there is a small decrease in performance incurred after calibration (moving the Let's Go! model into the RoomLine domain and moving the RoomLine model into the Let's Go! Public domain); in all other cases, performance improves. The calibration procedure significantly improves the performance in the transfer of the Let's Go! Public model (this model was the worst performing one in the un-calibrated setting.) When transferring into the RoomLine domain, the calibrated model closes now 41% of the gap between the majority baseline and the fully-supervised model (as opposed to 30% for the un-calibrated model). The improvement is larger when transferring to the Let's Go! domain: the calibrated model closes 67% of the gap (as opposed to 20% for the un-calibrated model.) All calibrated models perform better than the 100 data-points baseline, indicating that we can make more efficient use of small amounts of la-

beled data when an out-of-domain annotator is already present. Finally, while the `full`-calibration models perform better than the `small`-calibration models across the board, the differences between these models are relatively small indicating that we can get most of the potential recalibration gains by using this relatively small amount of labeled data.

5.4.3 Concluding remarks

We have performed a comparative analysis of four supervised learning approaches for training confidence annotation models (logistic regression, decision trees, boosting and Naïve Bayes), using data from three different spoken dialog systems. In a previous, more limited study with the CMU Communicator system, we have investigated two additional learning techniques: support vector machines and Bayesian networks [22]. Overall, several interesting observations emerged.

First, we have discussed the use of proper scoring rules (e.g. Brier score, log-likelihood) for evaluating confidence annotation models. In contrast to the widely used classification-error or accuracy metrics, proper scoring rules capture both refinement and calibration, two important properties for a confidence annotator. In fact, we have empirically shown that judging the performance of a confidence annotator based on classification-error alone can lead to misleading conclusions (see subsection 5.4.2.5.)

Second, we have seen that, after calibration, all the proposed supervised learning techniques with the exception of Naïve Bayes led to similar results. We suspect that the Naïve Bayes models are negatively impacted by the lack of independence between the features available for training the models. Perhaps in combination with a feature selection algorithm, this technique would lead to similar results. The constructed models have revealed that features extracted from different knowledge sources in the system (e.g. speech recognition, language understanding, dialog state and history) can provide useful and orthogonal information for the confidence annotation task. Performance stems mostly from the set of features used, rather than from the supervised learning technique.

The third conclusion regards the sample efficiency of the proposed models. Although once calibrated the supervised learning techniques we investigated led to similar results, the logistic regression models are the most sample efficient. When little training data is available, they significantly outperform the other models. Furthermore, no post-calibration procedure is required in their case. If we also take into account the simplicity of these models (both in terms of training and run-time use), we conclude that the logistic regression models are the most appropriate ones for the task at hand.

Finally, we have also investigated how well the confidence annotation models generalize across domains. Our empirical results indicate that, while some confidence annotation models might generalize well, this is not always the case. Furthermore, the transfer can be asymmetric: a model trained in domain A might generalize well to domain B, but not the other way around. We believe the explanation lies in the nature of the supervised-learning process. Data-driven confidence annotation models are automatically tuned to the characteristics of the dataset on which they were trained, and, unless the new domain has the same characteristics, an out-of-domain model will not perform as well as a model trained with in-domain data. We have shown that a simple calibration procedure using small amounts of labeled data from the target domain can be used to adapt an out-of-domain model into a new domain, generally leading to further performance improvements.

5.5 An implicitly supervised learning approach

Supervised learning approaches, such as the ones discussed in the previous section, have at least two important limitations. First, they require a pre-existing a corpus of in-domain[16] user utterances. Un-

[16] In the previous section, we have seen that confidence annotation models trained with out-of-domain data can give some performance gains, but fall short of models trained with in-domain data. Moreover, the type of cross-domain transfer described in the previous section makes the assumption that there is a common input-line infrastructure (i.e. the systems need to use the same recognition, understanding and dialog management components); this is not always the case.

fortunately, such corpora are difficult and expensive to collect and label, especially in the early stages of system development. Secondly, supervised learning techniques generally favor an off-line, or batch approach. A corpus is collected, manually labeled, and then the confidence annotation model parameters are estimated from this data. The resulting model mirrors the properties of the training corpus, but does not respond well to changes in the system's environment and the underlying distribution of data. Unfortunately, such changes are to be expected during the early stages of deployment. Oftentimes, the system authors might wish to alter certain aspects of system functionality based on early feedback and observations. In addition, as users repeatedly interact with the system and familiarize themselves with it, their behavior also changes (e.g. novice users become expert users.) Finally, the very introduction of a new confidence annotation model will in fact lead to changes in the interaction. Conversational spoken language interfaces are interactive systems that operate in dynamic environments, and shifts in the underlying distribution of the data are inevitable.

In this section, we propose and evaluate a novel approach for learning confidence annotation models that addresses these drawbacks. The proposed approach, **(online) implicitly supervised learning**, builds on a key property of spoken dialog systems: their interactivity. The central idea is to extract the required supervision signal from naturally-occurring patterns in the conversation, for instance from user corrections. In this approach, no developer supervision is required; rather, the system learns online, throughout its lifetime, by interacting with its users. We believe this new learning paradigm can be applied in a number of other problems, and represents an important step towards building routinely self-improving systems.

We begin by describing the details the proposed implicitly supervised approach, as applied to the confidence annotation problem, in the next subsection. Then, in subsection 5.5.2 we present empirical results from a set of initial experiments using this approach in two different dialog domains (RoomLine and Let's Go! Public). Finally, in subsection 5.5.3, we discuss several ideas for continuing this work and for applying the proposed implicitly-supervised learning paradigm to other problems.

5.5.1 Method

We start from the observation that a spoken dialog system can automatically obtain the labels necessary for building a confidence annotation model by leveraging a certain interaction pattern that occurs naturally in conversation. Consider the first example from Figure 67, extracted from the Let's Go! Public system. In the first turn, the system asked for the departure location. The user responded *"the airport"*, but this was misrecognized as "`Liberty and Wood`". Next, in turn 2, the system engaged in an explicit confirmation, trying to verify the departure location it heard (Liberty and Wood). The user, who is cooperative and invested in the interaction, corrected the system by answering *"no"*. The immediate reason for the system's action and user response in turn 2 was to allow the conversation to proceed correctly. Notice however that this interaction pattern – a user response to a system confirmation action – generates additional useful information. The system knows now that it misunderstood the user in the first turn, and it can use this information to train or refine a confidence annotation model.

Notice that the labels implicitly obtained via this interaction pattern are not noise-free. In the example discussed above, the user response was a simple "no", which was correctly understood by the system. In general, user responses to explicit confirmation actions extend beyond simple yes and no answers, and can also be subject to recognition errors (see [63], as well as our own empirical analysis from the next chapter, subsection 6.4.3.) As a consequence, the labels produced by this interaction pattern will not always be perfect. For instance, in the second example from Figure 67, the user response to the system's explicit confirmation action is also misunderstood ("no" instead of "yeah"), and this leads to an incorrect label.

This interaction pattern can therefore generate confidence annotation labels at a certain level of **accuracy**, and also at a certain **recall**. In this context, by **accuracy** we refer to the accuracy of the implicit labels with respect to the reference set of manually created labels. By **recall** we refer to the proportion of utterances for which this interaction pattern can generate labels (i.e. the utterances fol-

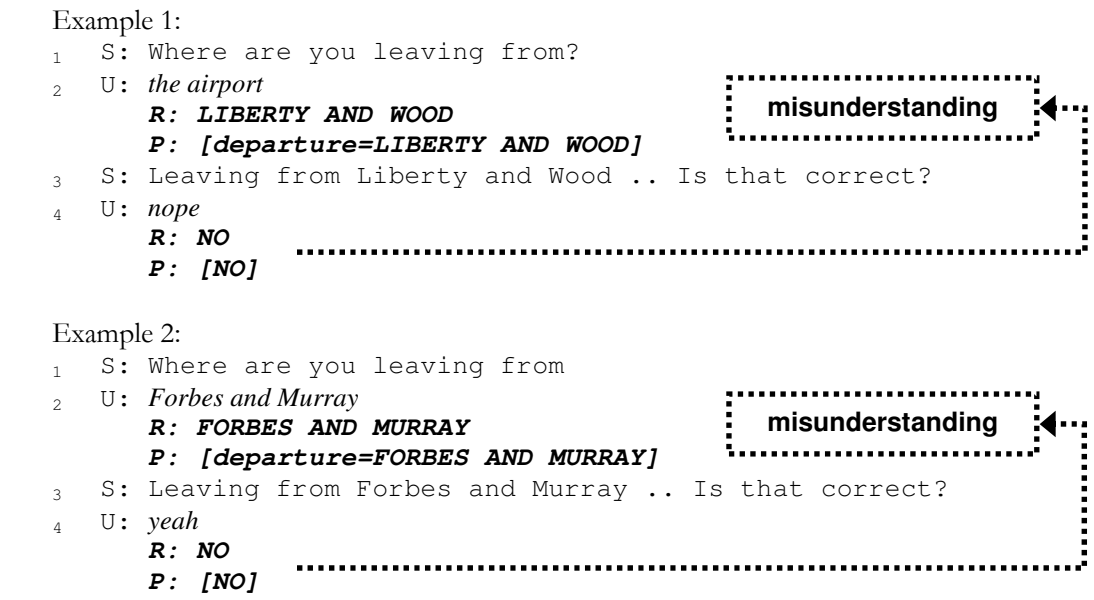


Figure 67. Leveraging explicit confirmation patterns for implicit learning of confidence annotation models (`S:` marks the system turns, `U:` marks the user turns, `R:` marks the recognition result, `P:` marks the semantic representation of the recognition result)

lowed by an explicit confirmation and a simple response.) The empirical analysis of user responses to explicit confirmation actions that we will describe in the next chapter indicates that a significant proportion of user responses are indeed as simple as *"yes"*, *"no"*, or equivalents (e.g. *"that's correct"*, *"wrong"*, etc.). In these cases, they are generally easy to recognize. As a consequence, the accuracy of the implicit labels should be high enough. If a system engages in explicit confirmations often enough, it should be able to collect sufficient reliable data to automatically train a confidence annotation model, without any explicit supervision from its developers.

In the implicitly supervised learning paradigm, the model training methodology remains the same, i.e. supervised learning. The labels are however obtained automatically, through interaction with the users. They can be used to constantly evaluate and adjust the confidence annotation model throughout the system's lifetime. For instance, a spoken dialog system could start by explicitly confirming all the information it acquires from the user – some systems do this routinely. As the system collects more labels through interaction and updates its confidence annotation model, its error detection abilities improve. The system can start trusting the confidence scores more, and use explicit confirmations only when these scores are very low.

To fully exploit the proposed implicitly-supervised learning paradigm, we need to better understand its properties, advantages and limitations. Three interesting questions arise: (1) can we make effective use of the labels obtained through interaction? (2) how can a system balance its long-term learning goals with the short-term need to efficiently provide information to the user? and (3) could a system discover new interaction patterns that can provide labels for confidence annotation? In this work, we have focused on the first one of these issues. In the next subsection, we present results obtained using the proposed implicitly-supervised learning methodology in two different dialog domains: RoomLine and Let's Go! Public. Then, in subsection 5.5.3, we briefly discuss the second and the third question outlined above. We believe the answers to these questions can lead the way towards building autonomously self-improving systems.

5.5.2 Experimental results in the RoomLine and Let's Go! Public domains

We conducted a set of initial experiments to evaluate the proposed implicitly-supervised learning methodology using corpora collected from two different dialog systems: RoomLine and Let's Go! Public. We did not use the Let's Go! corpus since that system did not engage in explicit confirmations, a pre-requisite for the proposed approach. Although the implicitly-supervised methodology enables an online approach to learning confidence annotation models, the experiments we describe below have been conducted in an offline fashion. We have focused our attention on the first one of the three questions outlined in the previous subsection: can we make effective use of the labels obtained through interaction? The next step, an online solution, remains as future work and is briefly discussed in subsection 5.5.3.

5.5.2.1 Implicit supervision signal

We have already outlined a naturally-occurring interaction pattern that can provide implicit labels for the confidence annotation task: simple user responses to the system's explicit confirmation actions. More precisely, in this work, we generated the implicit supervision labels as follows:

- if (1) the current dialog state is an explicit confirmation, and (2) the user response contains a positive confirmation marker (e.g. yes, right, correct, etc.) and (3) the previous dialog state is **not** an explicit confirmation, then label the previous user turn as correctly understood by the system;
- otherwise if (1) the current dialog state is an explicit confirmation, and (2) the user response contains a negative confirmation marker (e.g. no, wrong, incorrect, etc.) and (3) the previous dialog state is **not** an explicit confirmation, then label the previous user turn as incorrectly understood by the system;
- otherwise do not label the previous turn.

Condition number (3) ensures that the current explicit confirmation and the subsequent positive or negative user response refer indeed to the immediately previous user turn, rather than an earlier one. It is possible that a number of non-understandings occur following an explicit confirmation. This will typically lead to a sequence of explicit confirmation states, such as in Figure 68. In this case a positive or negative user response to the last explicit confirmation does not tell us anything about the previous user turn; rather, the user response corresponds to the initial turn that caused the system to engage in an explicit confirmation.

```
1  S: Where are you leaving from?
2  U: the airport
      R: LIBERTY AND WOOD
      P: [departure=LIBERTY AND WOOD]
3  S: Leaving from Liberty and Wood .. Is that correct?
4  U: no
      R: [NOW]
      P: NON-UNDERSTANDING
5  S: Leaving from Liberty and Wood .. Is that correct?
6  U: no
      R: [NOW]
      P: NON-UNDERSTANDING
7  S: Leaving from Liberty and Wood .. Is that correct?
8  U: no
      R: [NO]
      P: [NO]
```

Figure 68. Sequence of explicit confirmations with no implicit confidence annotation labels generated (`S:` marks the system turns, `U:` marks the user turns, `R:` marks the recognition result, `P:` marks the semantic representation of the recognition result)

5.5.2.2 Data

The experiments described below were conducted based on the RoomLine and Let's Go! Public corpora described previously in subsection 5.4.2.1. Recall that for these corpora we do have the reference (manually generated) confidence annotation labels.

The RoomLine and Let's Go! Public systems used very different policies for engaging in explicit confirmations. The RoomLine system made the decision based on the confidence score of the recognized utterance (at the time of the data collection, RoomLine was using a pre-existing confidence annotation model that had been trained on data from the CMU Communicator system [22].) RoomLine used a threshold-based, exploratory policy for engaging in confirmation actions: if the confidence score was below 0.3, the utterance was rejected. If the confidence score was above 0.3, then with probability 0.2, the system chose randomly between accepting the utterance, engaging in an implicit confirmation or engaging in an explicit confirmation; alternatively, with probability 0.8, the system followed a threshold-based policy: explicitly confirm if the confidence score is below 0.5, implicitly confirm if the confidence score is between 0.5 and 0.8, and accept if the confidence score is above 0.8. As a result, the total number of explicit confirmations in the RoomLine corpus is 1412, amounting to 17.6% of the total number of utterances (8037). These statistics are summarized in Table 21.

In contrast, given the more adverse environment, the Let's Go! Public system used a simpler, more conservative confirmation policy: the system always explicitly confirmed every piece of information received from the user (unless the confidence score was below 0.3, in which case the utterance was rejected.) The number of explicit confirmations in the Let's Go! Public corpus is therefore significantly larger – 2594, representing 43.0% of the total number of utterances (6029).

As a consequence of the different confirmation policies, the recall and also the accuracy of the implicit labeling scheme described above was different across the two domains. As expected,

Statistics	RoomLine	Let's Go Public
Total # of utterances	8037	6029
Total # of explicit confirmations	1412	2594
% of explicit confirmations	17.6%	43.0%
Total # Implicit labels	976	1998
Implicit labels recall	10.8%	33.1%
Implicit labels accuracy	89.9%	82.5%

Table 21. Implicitly supervised learning - corpus statistics

given that explicit confirmations were more often engaged in the Let's Go! Public system, the recall of the implicit labeling scheme was significantly larger than in the RoomLine system 33.1% versus 10.8% (see also Table 21.) At the same time, given the more adverse noise conditions and worse recognition performance in this domain, the accuracy of the labels is lower: 82.5% versus 89.9% in the RoomLine system.

5.5.2.3 Features

The same set of features described in subsection 5.4.2.3 and Table 15 was used in these implicitly-supervised learning experiments.

5.5.2.4 Supervised learning techniques

Given our previous observations with respect to performance and sample efficiency, we used logistic regression [76] as the supervised learning technique of choice in these experiments.

5.5.2.5 Experimental results

The confidence annotation models were trained using the same 20-fold cross validation procedure described earlier in subsection 5.4.2.5. The difference is that this time we only used the implicitly la-

beled portion of each training set. Because the number of implicitly labeled training points in the Let's Go! Public domain is larger and enables a more robust analysis, we begin by describing the results on this corpus domain

§ Experimental results in the Let's Go! Public domain

The average test-set performance (Brier score) of the implicitly-supervised confidence annotation model is illustrated in Figure 69. As this figure shows, the proposed implicitly-supervised approach has a Brier score of 0.1443, closing 75% of the gap between the majority baseline and the fully supervised model. Furthermore, a post-calibration procedure based on 100 randomly chosen labeled data-points further increases the model's performance to 0.1390, closing 80% of that gap. The difference between the un-calibrated and calibrated model is statistically significant (paired t-test, $p=0.002$). The proposed implicitly-supervised learning approach produces results that are close to the fully supervised model, without requiring any manual annotations.

The remaining performance gap between the implicitly- and fully-supervised models is explained by the smaller training set used in the implicit approach, i.e. the recall factor of the implicit labeling scheme, and by the noise in the implicit labels, i.e. the accuracy factor of the implicit learning scheme. To better understand the effect of these two factors on model performance, we constructed a number of additional models.

First, to distinguish between the effects of accuracy and recall, we constructed a model, dubbed **full-accuracy/same-recall (FA/SR)**. This model used for training only utterances that were implicitly labeled (hence same-recall), but in conjunction with the reference labels (hence full accuracy). The average test-set Brier score for this model was 0.1321, about half-way between the implicitly-supervised and fully-supervised models, with both differences statistically significant ($p<10^{-6}$) – see Figure 70. This result indicates that both the lack of recall, i.e. the smaller number of available training points, and the lack of accuracy in the implicit labels contribute in roughly equal amounts to the observed performance gap.

So far we have characterized the labels implicitly generated by user responses to explicit confirmations in terms of their accuracy and recall. Another factor plays however an important role: the sampling bias. It is important to notice that, even though the proposed interaction pattern provides labels for 33% of the utterances in the corpus, these 33% of the utterances are not randomly selected. Rather, these are utterances followed by explicit confirmations, which in turn are followed by simple user responses. The underlying distribution of the features in this subset of utterances does not necessarily match the general distribution in the full set of utterances. Similarly, the implicit labeling scheme might also bias the target labels (i.e. their accuracy) in a certain direction. For instance, if

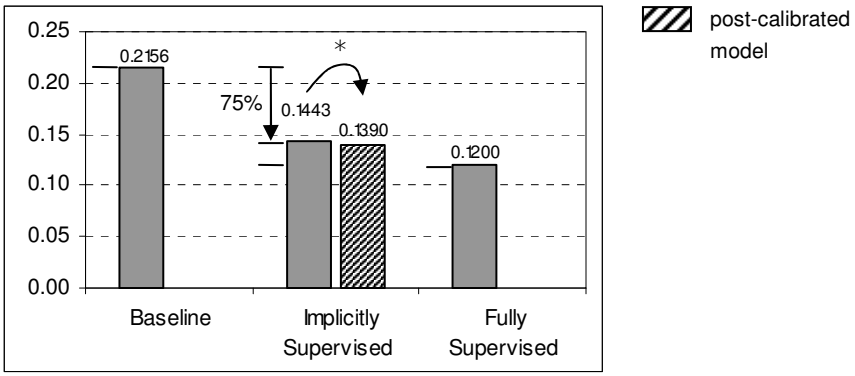


Figure 69. Implicitly versus fully-supervised learning approach on Let's Go! Public data; * indicates that the post-calibrated implicitly supervised model performs statistically significantly better than the uncalibrated model; the implicitly supervised model closes 75% of the performance gap between the baseline and fully supervised model

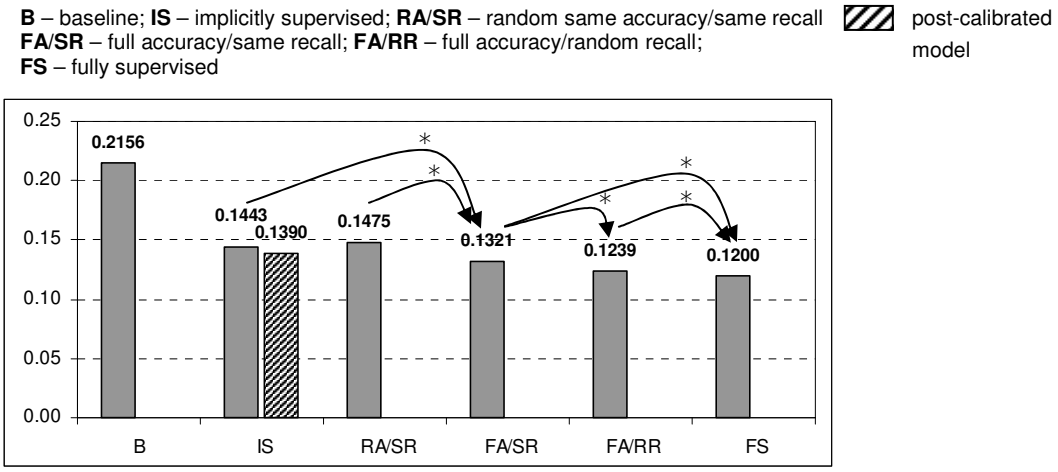


Figure 70. Implicitly- versus fully-supervised learning performance gap decomposition (arrows with stars mark statistically significant differences, $p<0.001$)

yes-type responses are more easily recognized than no-type responses, the implicit labels will be biased towards the "correctly-understood" class.

To investigate the effect of sampling bias on performance, we constructed two additional models. The first one of these models, **full-accuracy/random-same-recall (FA/RR)**, addresses the recall-bias issue and was trained with a randomly selected subset of utterances that has the same recall (size) as the implicitly labeled subset (hence random-same-recall). The second model, **random-same-accuracy/same-recall (RA/SR)**, addresses the accuracy-bias issue. This model uses the utterances that were implicitly labeled (hence same-recall); the training labels were however constructed by starting from the reference labels and randomly altering them to attain the same accuracy as the implicit labels. The test-set average Brier score performance of these models is also shown in Figure 70.

The performance of the full-accuracy/random-same-recall model, 0.1239, places it closer to the the fully-supervised model (0.1200) than to the full-accuracy/same-recall-model (0.1321). Both differences are statistically significant in a paired t-test ($p=0.002$ and $p=0.0002$ respectively). The larger difference to the full-accuracy/same-recall model seems to indicate that the recall bias does affect performance in this case. On the other hand, the random-same-accuracy/same-recall model performs similarly to the implicitly supervised model, in fact slightly worse (0.1475 versus 0.1443). No statistically significant difference can be detected between these two models. This result indicates that, at least in this domain, the proposed implicitly generated labels do not exhibit a detrimental accuracy bias.

On a final note, recall that in Figure 69 we have seen that the implicitly-supervised approach closes 75% of the gap between the majority baseline and a fully-supervised approach. A comparison with the full-accuracy/random-same-recall model is perhaps more informative, because these two models use the same amounts of labeled data. Correcting for sample bias represents a difficult and interesting research problem [140]. At the same time, we can easily envision using more data (since we don't need to manually label it). As more data becomes available, the full-accuracy/random-same-recall model will eventually reach the performance of the fully supervised model (given the performance asymptotes we have seen in subsection 5.4.2.5.) When compared to this model, the proposed implicitly-supervised approach closes 78% of the performance gap; the post-calibrated model closes 84% of this gap.

§

§

§ Experimental results in the RoomLine domain

We now shift our attention to the RoomLine domain. Here, due to the more optimistic confirmation policy, the recall of the proposed implicit labeling scheme is lower – 10.8%. At the same time, due to better environmental conditions and less speech recognition errors, the accuracy is higher – 89.9%.

The results in this domain are illustrated in Figure 71. Again, the implicitly-supervised approach attains a significant improvement over the majority baseline. The improvement is comparatively smaller with the one attained in the Let's Go! Public domain. On the RoomLine corpus, the implicitly supervised approach closes only 48% of the gap to the fully-supervised model; the post-calibrated model performs slightly better, but the improvement is not statistically significant. When compared to the full-accuracy/random-same-recall model, the implicitly supervised approach closes 59% of the gap. The inferior performance on the RoomLine domain was somewhat expected due to the more optimistic confirmation policy and the resulting lower recall of the implicit labeling scheme. In the ideal case, in order to build a confidence annotation model using implicit learning we would like the system to start with an always-confirm policy, like Let's Go! Public system did. Overall, in the RoomLine corpus we had only 977 implicitly labeled training points, while in the Let's Go! corpus we had more than double. The full-accuracy/same-recall model (FA/SR in Figure 71), confirms that a significant part of the remaining performance gap is indeed explained by the lower recall. At the same time, a significant amount of the remaining performance gap is explained by the lack of accuracy. This is somewhat surprising, since the accuracy is higher than in the Let's Go! Public domain. A possible explanation is that, when only small amounts of data are available for training, and/or when the class marginals are more skewed[17], precision plays a more important role. Finally, the random-same-accuracy/same-recall and full-accuracy/random-same-recall models reveal that there is no detrimental sampling or recall bias in this domain. Like before, as the amount of training data increases, we can expect the gap between the full-accuracy/same-random-recall and fully-supervised model to decrease; further performance gains for the implicitly-supervised model are therefore expected, as we increase the dataset size.

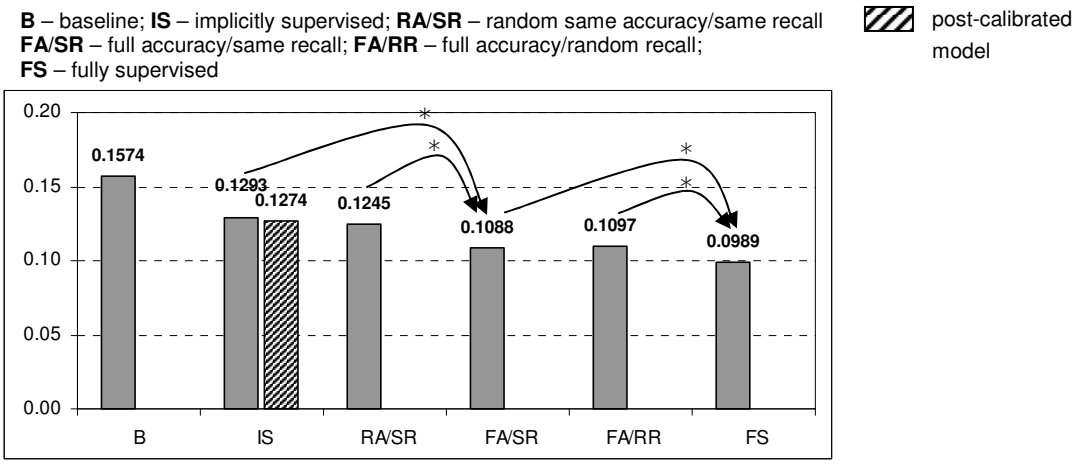


Figure 71. Implicitly- versus fully-supervised learning in the RoomLine domain (arrows with stars mark statistically significant differences, $p<0.001$)

§ Performance versus training set size

To better understand these underlying trends, we also investigated the relationship between the performance of the implicitly-supervised confidence annotation models and the overall training set size.

[17] The majority baseline is 19.6% in the RoomLine corpus and 31.4% in the Let's Go! Public corpus.

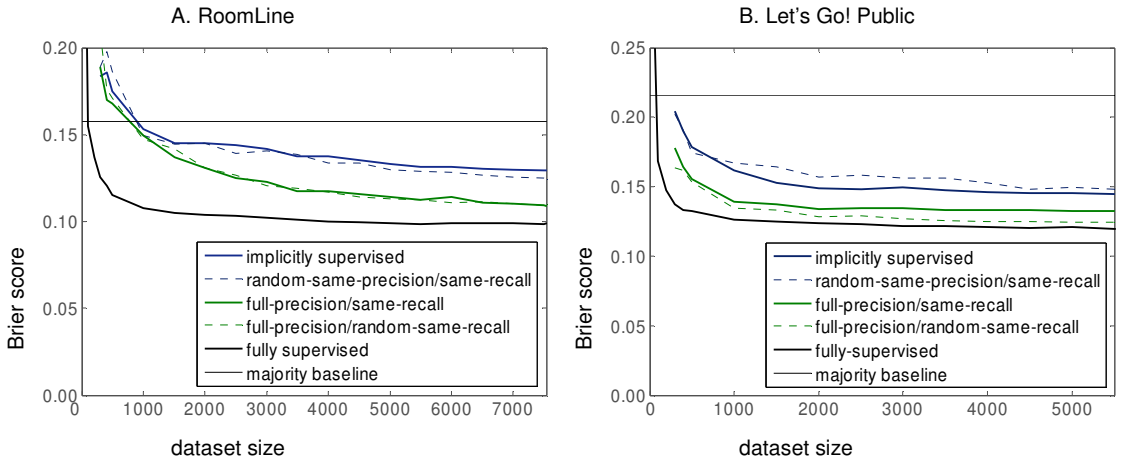


Figure 72. Implicitly supervised confidence annotation model performance as a function of training set size (in the RoomLine and Let's Go! Public domains)

The results are shown in Figure 72.A for the RoomLine domain, and Figure 72.B for the Let's Go! Public domain.

In the RoomLine domain, the performance of the implicitly-supervised model does not yet reach an asymptote by the time we have considered the full training set (7537 utterances.) This result corroborates our previous conjecture that, if more data were available, further performance gains would be possible. As more data becomes available, the full-precision/random-same-recall model is guaranteed to reach the same asymptote as the fully supervised model. At the same time, we expect that the gap between the implicitly supervised method and the full-precision/random-same-recall model will stay roughly constant. As a consequence, we expect corresponding gains in the implicitly-supervised model performance.

Another interesting observation is that the random-same-precision/same-recall model closely tracks the implicitly supervised model, and the full-precision/random-same-recall model closely tracks the full-precision/same-recall model. These trends confirm that there is no detrimental sample bias (neither in terms of accuracy nor recall) in the proposed implicit learning scheme in the RoomLine data.

In the Let's Go! Public domain, the implicitly-supervised model seems to have reached a performance asymptote; this is not surprising, given the larger recall of the implicit labeling scheme in this domain. As the amount of data increases, the full-precision/random-same-recall model shows increasingly larger improvements over the full-precision/same-recall model.

5.5.3 Concluding remarks

In this section, we have proposed a novel paradigm for constructing confidence annotation models in conversational spoken language interfaces: implicitly-supervised learning. In contrast to previous supervised learning solutions, the proposed approach does not require developer supervision. Instead, the system obtains the supervision signal directly from its users, from their responses to the system's explicit confirmation actions. In effect, the system learns from its own experiences.

To evaluate the proposed approach we have conducted a set of initial experiments in two different dialog domains: RoomLine and Let's Go! Public. The experiments were centered on assessing (1) various characteristics (e.g. precision, recall, bias) of the confidence annotation labels that can be obtained through interaction, and (2) how these characteristics affect the model building process. The empirical results confirm that a system can indeed successfully leverage interaction patterns to automatically construct a confidence annotation model that performs similarly to a fully-supervised model. In the Let's Go! Public domain, the proposed implicitly-supervised learning approach closed 78% of the performance gap between a majority baseline and a supervised model (84% if an addi-

tional 100 manually labeled data-points are available for calibration.) In the RoomLine domain, the proposed approach closed 59% of the performance gap to the supervised model, but we expect that further improvements could be made if more data unlabeled were available.

While these initial results are very encouraging, they represent only a first step towards understanding the properties, advantages and limitations of the proposed implicitly-supervised paradigm. So far, we have performed a batch mode evaluation. However, apart from eliminating the requirement for a manually labeled corpus, a second important advantage of the implicitly-supervised paradigm is that it facilitates online learning and adaptation. The next question therefore is: **how can a system elicit knowledge-producing interaction patterns in pursuit of its learning goals but without significantly disrupting the interaction?** This is essentially an online control problem, where the system must balance the benefits of gaining knowledge by engaging in explicit confirmations with the costs potentially incurred by the user. If used excessively, explicit confirmations can have a negative impact on the quality of the interaction, and lead to user frustration.

To a certain extent, dialog managers already have to solve similar trade-offs when deciding between different confirmation strategies, such as explicit or implicit confirmation. Explicit confirmations take an extra dialog turn, but the system has a better chance of understanding the follow-up user response, especially if the information to be confirmed is incorrect (see [63], as well as our own analysis from the next chapter, subsection 6.4.3.) Typically, the dialog costs are assumed to be known and are immediate. Solutions to these trade-off problems range from hard-coded heuristics [61, 91] to various offline corpus-based methods [36, 114, 136]. In an online implicitly-supervised approach, the additional learning goals change the nature of the problem in two different ways. First, system actions not only create immediate dialog costs, but also produce knowledge that can be used to improve future performance. To address this new trade-off, the system must be able to assess the long-term benefits of the knowledge that stands to be gained: given the current state, what is the utility of one additional data point (at a certain expected precision and bias) for future system performance? Secondly, we would like an online solution for this problem. Systems should be able to continuously monitor their current performance and adjust their control policies online, as their models improve.

Finally, another interesting question regards the knowledge-producing interaction pattern itself. In the experiments discussed above, the knowledge-producing interaction pattern consisted of user responses to system confirmation queries. Intuitively, other informative patterns could be found. For instance, if in a certain segment the dialog advances normally towards its goals (i.e. the dialog-state progression looks normal) and no non-understandings happen, we might consider all those user turns correctly understood by the system; in this case we obtain positive labels for confidence annotation. Alternatively, if a certain concept is corrected by the user at a later point in the dialog, we might mark the initial utterance from which the system extracted the first value for that concept as incorrect; in this case we obtain negative labels. We believe that an interesting avenue for future research is to **develop techniques that allow systems to automatically discover such knowledge-producing interaction patterns**.

A possible approach to this problem is to cast pattern discovery as a feature selection problem in a supervised learning task. Assume that we do have a small, pre-existing labeled corpus. We can define a set of features corresponding to each user turn by including not only attributes of that particular turn, but also attributes of prior and subsequent user and system turns. Interaction patterns can therefore be captured by various combinations of these features. For instance, the explicit confirmation pattern we have used so far can be captured as follows: `(system_action_in_next_turn=explicit_confirmation) AND (user_response_in_next_turn=yes)`. Once an appropriate set of features is established, we can investigate various supervised learning techniques in conjunction with feature selection methods to detect other interaction patterns (i.e. feature combinations) that correlate well with our target labels. An interesting challenge from a machine learning perspective will be dealing with the very large feature space (each user response in our systems can be characterized by several hundred features extracted from different knowledge sources, e.g. acoustic, prosodic, language, understanding, and dialog). We believe that, with a carefully designed feature set, this proposed method can lead to the discovery of new and interesting patterns,

or of useful improvements to existing patterns. In effect, the system will mine the pre-existing labeled corpus to discover interaction patterns that correlate well with confidence labels (note that the patterns can reference features from "future" turns, and therefore cannot be used directly as a runtime confidence annotation model). Next, the system can use the patterns it has discovered to automatically label a larger, unlabeled corpus of current experiences. In a final learning step, the system learns a runtime confidence annotation model using only features available at runtime from this automatically labeled corpus.

We believe that the proposed implicitly-supervised learning paradigm can be applied to a number of other learning problems, both in conversational spoken language interfaces, and in the more general class of interactive systems. The central idea is to acquire knowledge online, by discovering, eliciting and leveraging natural patterns that occur in interaction as a by-product of the collaboration between the system and an invested user. This paradigm can eliminate the need for developer supervision and enables fast online adaptation and learning. We conjecture that it can supplement and or even provide a strong alternative to existing learning approaches, and enable significant autonomous learning in interactive systems.

5.6 Summary and future directions

In this chapter, we have investigated different methodologies for building semantic confidence annotation models in the context of conversational spoken language interfaces.

Semantic confidence annotation can be viewed as a pattern recognition, or detection task: given a set of features that characterize a semantic hypothesis for the user's turn, determine whether or not this hypothesis corresponds to the user's expressed intent. We began by investigating the use of supervised learning techniques for this task, and centered our attention on a number of issues less thoroughly investigated in the literature: evaluation metrics, sample efficiency, and how well the learned models generalize across domains.

Specifically, we performed a comparative analysis of four different supervised learning techniques (i.e. logistic regression, boosting, decision trees and Naïve Bayes), using data from three different dialog domains. The analysis indicates that different knowledge sources in the system can generate informative features for the confidence annotation process. Additionally, the analysis highlights the importance of using proper probabilistic scoring rules (e.g. log-loss or Brier Score) when evaluating the performance of confidence annotation models. In fact, empirical results show that an evaluation that relies solely on classification error (a commonly used metric) can lead to misleading conclusions. In terms of performance, we have seen that once the outputs are calibrated, different supervised learning techniques perform similarly. At the same time, logistic regression models directly produce well-calibrated confidence scores, and are more sample efficient: when only small amounts of training data are available, logistic regression significantly outperforms the other supervised learning techniques. We have also analyzed how well confidence annotation models generalize across domains. Results indicate that while some confidence annotation models transfer well across domains, this is not always the case and the transfer can be asymmetric. Post-transfer calibration with small amounts of labeled data from the target domain can bring further improvements.

The analysis described in this chapter also points to a number of interesting directions for future research with respect to the use of supervised learning for confidence annotation. First, we have seen that a number of features from different knowledge sources in the system provide useful and complementary information for the confidence annotation task. We believe that the key for further improving confidence annotation performance lies with identifying additional knowledge sources and additional features. Second, the ability to transfer a trained confidence annotation model into a new domain is very important; training a new model for each new domain is a labor-intensive process that we would like to shortcut. The cross-domain analysis we have described above, although to our knowledge is the first one of this kind in dialog systems, was limited to the three available corpora. In the future, it would be interesting to extend this analysis to other corpora in an effort to gain a better understanding of the transfer process. An interesting direction of research would to investi-

gating the distributional similarities between the corpora and to identify (without labels in the new domain) the conditions under which we can expect an existing confidence annotation model to generalize to a new domain. A second interesting direction of research is the use of active learning techniques for post-transfer calibration. In the work described in this chapter, we have proposed a very simple calibration method that relies on a small random sample of labeled data-points in the target domain. An active-learning approach (in which we identify the sample to be labeled based on the existsing model) might lead to further performance gains.

In the second part of this chapter, we have proposed a novel paradigm for building confidence annotation models that addresses two limitations of the classical supervised learning approach: the need for a pre-existent manually labeled corpus of utterances and the batch-style model building process. In the new approach, dubbed implicitly-supervised learning, no explicit developer supervision is required: the system acquires the labels necessary for training the confidence annotation model through interaction, from user responses to the system's explicit confirmation actions. Results from a set of initial experiments with data from two different dialog domains have confirmed the feasibility of the proposed approach.

The experiments we have reported here represent only a first step towards a fuller understanding of the proposed implicit-learning paradigm. The encouraging results we have obtained on the confidence annotation task point towards what we believe to be a very interesting research avenue. We conjecture that the proposed approach can be applied to address a number of other problems in conversational spoken language interfaces, and in interactive systems in general. To fully exploit this approach, we first need to better understand its properties, advantages and limitations. Specific questions that will need to be addressed include: what are the characteristics (e.g. accuracy, recall distributional biases, etc.) of knowledge obtained through an interaction pattern? How sensitive are current modeling approaches to variations in the quality and quantity of knowledge available? Can we develop techniques for making effective use of the knowledge extracted through interaction? Can we develop techniques that allow systems to automatically monitor the knowledge they have, need, or can acquire? Can we compute the utility of incremental additions to knowledge? Can this information be used to actively manage the learning process? Can we develop techniques that automatically discover new knowledge-producing patterns in the interaction? We believe that answers to these questions will pave the way towards developing autonomous, adaptive and self-improving interactive systems.

Chapter 6

Belief updating

While confidence scores can provide an initial assessment for the reliability of the information obtained from the recognizer, ideally spoken language interfaces should continuously monitor and improve the accuracy of their beliefs throughout a conversation, by leveraging information available in subsequent user turns. In this chapter we formalize this belief updating problem and we propose and evaluate a scalable, data-driven solution. The proposed approach relies on a compressed, abstracted concept-level representation of beliefs and casts the belief updating problem as a multinomial regression task. Experimental results indicate that the constructed belief updating models significantly outperform typical heuristic rules used for this purpose in current systems. Furthermore, a user study with a mixed-initiative spoken dialog system shows that the proposed approach leads to significant improvements in both the effectiveness and the efficiency of the interaction across a wide range of recognition error rates.

6.1 Introduction

In the previous chapter, we have investigated supervised and unsupervised methods for constructing semantic confidence scores. These scores can be used by spoken dialog systems to assess the reliability of their inputs, on a turn-by-turn basis. Based on the confidence score, the system can construct an **initial** belief about the value of the concepts contained in the recognition hypothesis. However, conversation occurs over multiple turns, and subsequent user responses oftentimes also carry relevant information that could be used to change the system's initial beliefs. Ideally, spoken dialog systems should leverage this information, and continuously **update** and **improve the accuracy** of their beliefs, throughout the whole dialog.

For instance, consider the first example shown in Figure 73. Based on the user response from turn 2, the system constructs an initial belief that the `start_time` for the desired room reservation is 10 a.m., with probability 0.35. Since the probability is low, the system engages in an explicit confirmation to validate this belief in turn 3. The recognized user response is *"yes"*. Based on this response, the system can now update its belief about the `start_time` concept, presumably by boosting the probability that the `start_time` is indeed 10 a.m.

Although in this first example the belief update is seemingly trivial, this is not the case in general. A few problematic cases are illustrated in Figure 73. In the second example, the initial system belief is again that `start_time` is 10 a.m. with probability 0.35. Again the system engages in an ex-

Example 1:

```
                          1 S:  For what time do you need the room?
                          2 U:  10 a.m.
                            R:  TEN A_M / 0.35
start_time={10am/0.35}
                          3 S:  Did you say 10 a.m.?
                          4 U:  yes
                            R:  YES / 0.87
start_time={10am/?}
```

Example 2:

```
                          1 S:  For what time do you need the room?
                          2 U:  10 a.m.
                            R:  TEN A_M / 0.35
start_time={10am/0.35}
                          3 S:  Did you say 10 a.m.?
                          4 U:  yes until noon
                            R:  GUEST UNTIL ONE / 0.87
start_time={10am/?}
```

Example 3:

```
                          1 S:  For what time do you need the room?
                          2 U:  10 a.m.
                            R:  TWO P_M / 0.65
start_time={2pm/0.65}
                          3 S:  starting at 2p.m. ... until what time?
                          4 U:  I need a different time
                            R:  CAN YOU DETAILS TIME / NONU-0.00
start_time={2pm/?}
```

Example 4:

```
date={Mon/0.83}           1 S:  I found 3 rooms Monday between 2 and 4
                                p.m. Would you like a small one or a large
                                one?
                          2 U:  no not Monday, Friday
                            R:  ABOUT MONDAY THURSDAY / 0.22
date={Mon/? ; Th/?}
```

Figure 73. Sample 1-step belief updating problems in a conference room reservation system (`S:` marks the system turns, `U:` marks the user turns, `R:` marks the recognition results; system beliefs are shown on the left-hand side)

plicit confirmation. However, this time the user's response is misrecognized by the system as *"GUEST UNTIL ONE"*. How should the system update its belief in this case? Even for the first example, because there a small but non-negligible chance that *"YES"* was a misrecognition, what exactly should the updated probability be for `start_time`=10 a.m.? In the third example the system engages in an implicit confirmation. The initial belief is incorrect and the user attempts to correct the system by saying *"I need a different time"*. This response is unfortunately misrecognized as *"CAN YOU DETAILS TIME"*, which produces a non-understanding. Again, it is not clear how exactly the system should update its belief in light of this recognition result. Finally, in the last example, the system doesn't even engage in a confirmation action. It just presents the list with the rooms it found to the user. However, since the value for the date concept is incorrect, the user interjects a correction, but this correction is misunderstood by the system. How should the system take into account the perceived user response, and what should the system believe about the date for the reservation in light of this interaction?

In the sequel, we will refer to this problem as the **1-step belief updating problem**: how should systems update their beliefs in light of follow-up user responses to various system actions? Most spoken dialog systems rely on simple heuristic rules to perform these updates. In this chapter,

we argue that current heuristic approaches are suboptimal. Instead, we propose a novel data-driven, machine-learning based solution for the belief updating problem. The proposed approach bridges ideas from confidence annotation and correction detection into a unified framework that allows spoken dialog systems to integrate information across multiple turns in the dialog, and to continuously update and improve the accuracy of their beliefs.

The rest of this chapter is structured as follows: we begin by reviewing related work in the next section, 6.2. Then, in section 6.3 we formalize the belief updating problem and discuss evaluation criteria for this task. In section 6.4, we present in detail the proposed model and we report empirical results based on data collected with the RoomLine system. In section 6.5, we investigate the impact of the proposed belief updating mechanism on global dialog performance. Finally we present a number of concluding remarks and discuss several opportunities for further advancing this work in section 6.6.

6.2 Related work

6.2.1 Confidence annotation

Semantic confidence scores clearly play an important role in the belief updating process. They provide the basis for constructing the initial system beliefs, and can also be used to assess the reliability of the information present in the subsequent user turns. In the previous chapter, we have already reviewed the semantic confidence annotation literature and we have discussed in detail various methodologies for constructing these scores.

6.2.2 Correction detection

A second vein of previous research relevant for the belief updating problem is work on correction detection [54, 68, 69, 121]. If in semantic confidence annotation the goal was to detect whether or not a user turn was correctly understood by the system, in correction detection the goal is to detect whether or not a particular turn represents a user correction. For instance, turn 4 from example 3 and turn 2 from example 4 in Figure 73 are user corrections. In these cases an accurate correction detector might be able to signal to the dialog manager that the user is attempting to correct the system. The dialog manager could take this information into account to update its beliefs.

The methodology for building correction detectors is very similar to the one for building semantic confidence annotators. Typically the problem is cast as a binary classification task. User corrections are manually labeled in a corpus of dialogs, and machine learning techniques are then used in conjunction with a large number of features to derive a correction detection model. For instance, in [121], Swerts et al. show that corrections are different from non-corrections prosodically. Based on these findings, the authors then use a rule-based machine learning algorithm (RIPPER) in conjunction with prosody, recognition and dialog features to detect corrections [54]. The trained detector is able to identify corrections with a classification error rate of 16% (versus a 29% majority baseline). The study also indicates that, although prosody can be informative, simply using generic recognition features yields similar performance on the correction detection task. In [69], the same authors take this work a step further and aim to detect not only corrections, but also aware-sites. These are turns in which the user becomes aware for the first time that the system has committed a recognition error; the user might however not yet correct the system at this point. Their results indicate that aware-sites are also different prosodically and can be detected at a 12% error-rate. In earlier work [68], Levow showed that corrections of system misunderstandings are prosodically different from corrections of system non-understandings, and that a decision tree can be used to distinguish between these two classes of corrections, with an accuracy of about 75%.

Both confidence and correction scores carry relevant information for the belief updating task. However, taken in isolation, they do not provide a solution to the problem. For instance, even if an estimate for the probability of correction in turn 4 from example 3 were available, it would still be

unclear how exactly the system should update the confidence score for the `start_time` concept. We argue that an integrated approach is needed, in which information about potential misunderstandings and corrections is brought together in a unified framework for accurate belief updating.

6.2.3 Heuristic belief updating

In the examples from Figure 73, we have assumed that the dialog system can track multiple alternate values for a single concept – see example 4. This however is not the case in most spoken dialog systems. Usually, a single value is stored, together with the confidence score, or a confidence flag (e.g. low, medium, high confidence). If a new value is present in the user response, it will generally overwrite the old value. In general, simple heuristic rules are used to update the confidence scores.

For instance, after engaging in an explicit confirmation, most systems will look for a yes-type answer, such as *"yes"*, *"that's correct"*, *"right"*, etc., or a no-type answer, such as *"no"*, *"that's wrong"*, etc. If a yes-type answer is detected, the current hypothesis is considered confirmed or grounded, i.e. the confidence is set to 1.0, or high-confidence. Alternatively, if a no-type answer is detected, the current hypothesis is deleted altogether. In the case of implicit confirmations, most systems rely on the user to overwrite the concept if the confirmed value is incorrect.

We believe that these heuristic approaches are suboptimal for a number of reasons. First, they do not really take the initial and subsequent confidence scores into account. Secondly, as the third example in Figure 73 illustrates, users don't always overwrite previous concept values. Previous studies have revealed that user responses following implicit confirmations cover a wide language spectrum [132], and simple heuristic rules will fall short on that account. As we show in subsection 6.4.3, even in the cases of explicit confirmations, user responses are not limited to yes- and no-type answers. Furthermore, user responses to system confirmation actions are also subject to speech recognition errors. This renders the problem even more difficult for rule-based approaches (see examples 2, 3 and 4 in Figure 73). Finally, these heuristic rules lead to polarized beliefs, such as trust that value or don't trust that value. The resulting beliefs are not well calibrated, and cannot be reliably used in a decision theoretic approach for error handling. In light of the shallow solutions currently used for belief updating, results such as "users discovering errors through implicit confirmations are less likely to get back on track" [113] are not so surprising. The problem might not lie with the implicit confirmations per se, but rather with an inability to handle user responses and to accurately update beliefs.

6.2.4 Other related work

The idea of tracking multiple alternate hypotheses and updating beliefs through time appears in a few other previous works. In [57], Paek and Horvitz present DeepListener, a spoken language system that fuses evidence from multiple turns to guide clarification dialog using dynamic belief nets. The system operates in a simple command-and-control domain, where the goal is to detect the user's intention. The intention is captured with a single concept, which can take one of 5 different values. The authors handcraft a dynamic Bayesian network that uses information about context, the user action and the recognized hypothesis to infer the user's goal. The probability distribution (belief) over the user's intention is then used in conjunction with handcrafted utilities to engage in various clarification actions. In this chapter we take inspiration from Horvitz and Paek's work, and propose a belief updating approach that scales up to complex spoken language interfaces and does not require handcrafting a large set of parameters. The proposed belief updating model is automatically induced from labeled data. The model can be scaled up to systems that operate with an unlimited number of concepts (slots), which in turn can take an unlimited number of possible values. Last but not least, the proposed learning approach allows us to consider a very large number of features from different knowledge sources in the system. The approach does not require expert knowledge about the potential relationships between these features, as a more structured model would.

Previous work by Higashinaka et al [51] also bears some commonalities with the problem and approach discussed in this chapter. In [51], Higashinaka et al describe a data-driven method for

updating the dialog-state in a spoken dialog system. Their system tracks the likelihood of multiple alternate dialog states by using statistical information about the probabilities of various dialog-act/dialog-act and dialog-state/dialog-act sequences. Each dialog-act is assigned a score s_{act} based on the linguistic and acoustic scores from the recognizer. The dialog-act/dialog-act bigrams and dialog-state/dialog-act bigrams are also assigned scores based on the frequency of their occurrence in the training corpus – s_{ngram} and s_{col}. Then, the score for each next dialog state candidate is computed using the following heuristic: $S_{t+1} = S_t + \alpha \cdot s_{act} + \beta \cdot s_{ngram} + \gamma \cdot s_{col}$ (the α, β, and γ weighting factors are set manually). In contrast, in the work discussed here, we focus on updating beliefs about the concepts the system operates with, rather than over which state the system is in. In RavenClaw, the answer to the latter question is inferred automatically by the dialog engine, based on the beliefs about the individual concepts. The problem we are addressing is different (and arguably more difficult) since the space of possible values for a concept is much larger. Furthermore, we investigate a significantly larger feature set and provide an entirely data-driven solution for the problem.

6.3 Problem statement

We have already introduced the **1-step belief updating problem** in the previous section: how should the system update its beliefs in light of subsequent user responses to various system actions?

Let us formalize this problem. Let C denote a concept that the system acquires from the user (e.g. `start_time`, `date`, etc). By $B_t(C)$, or system belief over a concept C, we denote a representation of the system's uncertainty in the value of concept C at a certain time t. In the most general case, this would be a probability distribution over the full set of possible values for the concept C. Let SS_t be the system state at time t; let $SA_t(C)$ denote the system action with respect to concept C at time t; finally, let R_t denote the perceived user response to the system action at time t. Then, the 1-step belief updating problem can be restated as follows:

> given the system state at time t – SS_t, and the user response R_t to the last system action, compute the updated belief over concept C – $B_{t+1}(C)$.

$$B_{t+1}(C) \leftarrow f(SS_t, R_t)$$

Two components of the system state clearly play a very important role in the belief updating process: the initial belief at time t over concept C – $B_t(C)$, and the system action taken with respect to this concept at time t - $SA_t(C)$. Note that the system might take multiple actions with respect to different concepts in the same turn. For instance, in turn 3, example 3, Figure 73, the system implicitly confirms the `start_time` concept and requests the `end_time` concept. Therefore, with respect to the `start_time` concept, the system action can be described as:

SA_t(`start_time`) = ImplicitConfirm(`start_time`) + Request(`other`)

At the same time, with respect to the `end_time` concept, we have:

SA_t(`end_time`) = ImplicitConfirm(`other`) + Request(`end_time`)

If we consider only these two components of the system state in the belief updating process, the problem becomes:

> given an initial belief over a concept $B_t(C)$, the last system action with respect to this concept - $SA_t(C)$, and the user response to this action - R_t, compute the updated belief $B_{t+1}(C)$.

$$B_{t+1}(C) \leftarrow f(SS_t:\{B_t(C), SA_t(C)\}, R_t)$$

In formulating the problem this way, we have made a couple of assumptions. First, we assumed a Markovian belief updating process. In other words, the system's belief at time t+1 only depends on the system's belief at the previous time-step t. Secondly, we assumed concept-

independence. The updated belief for concept C only depends on the initial belief for that particular concept and the corresponding system action. Generally, this might not be the case, and concept interdependencies might exist; for instance if the user specifies the start time for a conference room reservation, this introduces constraints on the possible values for the end time for the reservation, i.e. the end time cannot precede the start time.

One way to relax these assumptions is to introduce an extra variable in the model. Let X_t denote any additional system state information that might be pertinent to the belief updating process. X_t can include information about concept histories, beliefs about other concepts, as well as other global or historical information about the dialog, for instance how well the dialog has been going so far, etc. Then, the problem can be stated as:

$$B_{t+1}(C) \leftarrow f\ (\ SS_t:\{B_t(C),\ SA_t(C),\ X_t\},\ R_t,\ )$$

Some of the belief updating models we discuss in the following sections incorporate such additional information.

6.3.1 Evaluation criteria

In the previous chapter, we have shown that evaluating semantic confidence annotation models in terms of classification accuracy is insufficient, and sometimes even misleading. Instead, we have argued that proper scoring rules that measure both calibration and refinement should be used (see section 5.3.1.) The same holds true for belief updating models: we are interested in generating beliefs that are both well-calibrated and refined. To provide a complete picture, we will therefore evaluate the performance of the proposed models both in terms of accuracy and Brier score.

6.4 A supervised learning approach for belief updating

In the previous sections, we have introduced and formalized the belief updating problem. In this section, we propose and evaluate a scalable, data-driven solution to this problem. We describe in more detail the proposed approach in the next subsection, 6.4.1. Then, in subsection 6.4.2 we describe the corpus that was used to develop the proposed belief updating models, and in subsection 6.4.3 present an analysis of user responses to system confirmation actions based on this corpus that sheds more light on the challenges we face in the 1-step belief updating problem. Next in subsections 6.4.4 through 6.4.6 we discuss a set of increasingly more complex belief updating models. We compare these models to each other in subsection 6.4.7. Finally, we conclude this section with a summary of our findings in subsection 6.4.8.

6.4.1 Method

We propose a supervised learning approach for constructing the one-step belief updating model: we will learn the function f from labeled data. To make the problem tractable, we use a compressed, abstracted representation of beliefs. We begin by discussing this representation in the next subsection. Then, in subsection 6.4.1.2 we discuss a refinement in this representation that allows us to dynamically add new hypotheses and drop old ones from this compressed belief space throughout the interaction. Finally, we present the proposed supervised learning model in the third subsection, 6.4.1.3.

6.4.1.1 A compressed, abstracted belief representation

So far, by belief over a concept, B(C), we have denoted a representation of the system's uncertainty in the value of that particular concept. But how are beliefs represented? In the most general case, the system's belief over a concept can be modeled by a multinomial probability distribution over the full set of possible values for that concept. Consider for instance, the `departure_city` concept in a spoken dialog system for flight reservations. This concept can be represented by a multinomial variable that can take one of several hundred possible values: {Aberdeen, Abileen, Albany, Albuquerque

… Yuma}. The system's belief about the value of this concept can be modeled by a multinomial distribution over this set of possible values (see left-hand side of Figure 74).

$$B(\texttt{departure\_city}) = \langle p_{Aberdeen}, p_{Abileen}, \ldots p_{Yuma} \rangle,$$

$$\text{where } p_{Aberdeen} + p_{Abileen} + \ldots + p_{Yuma} = 1.$$

This type of representation poses however several challenges. First, the concept cardinality can range from very small (yes/no or Boolean concepts) to large (~800 city names in the CMU Communicator system [101]) and very large (~20k song titles in a music library). This renders learning approaches intractable, due to data sparsity issues. Furthermore, a model using this representation would be specific to the actual space of possible concept values. Separate belief updating models would therefore be needed for each concept, and no generalization would be possible. Finally, this representation does not distinguish between the belief that all concept values are equiprobable, and the belief that the user has not yet specified any value for the concept – they would both be encoded by a uniform distribution.

To address these issues, we introduce a compressed and abstracted representation of beliefs. We start from the observation that, while concept cardinalities might be large, a speech recognizer will generally produce only a small number of different, conflicting values for a concept throughout a conversation. This intuition is indeed confirmed by empirical evidence. In a corpus collected with the RoomLine system, the system "heard" at most 3 different conflicting values for any given concept throughout any given conversation. Furthermore, the system "heard" more than one value for a concept in only 7% of the cases. We believe that, even if the system would extract and use more information from an n-best list (our system, like most other systems, did not), the number of alternative conflicting values for a concept that a system might hear throughout a conversation would still be a relatively small integer (3-7).

Under these circumstances, a compressed representation of beliefs has the potential of greatly simplifying the problem, without causing any significant degradation in performance. Instead of keeping a multinomial distribution over the full set of possible values for a concept, we abstract over the actual concept values, and keep the probabilities only for the **k** most likely values, where k is a small integer. We accumulate the rest of the probability mass into an **"other"** category. The identity of the k most likely values is stored separately, and, as we shall see later, might change from one time-step to the next. In the sequel, we use the term **k+other** to denote this compressed, abstracted belief representation.

Consider the example from Figure 74. The left-hand side illustrates a belief over the `departure_city` in a spoken dialog system that handles flight reservations. This concept can take one of n possible values: {Aberdeen, Abileen, Albany … Yuma}. The full belief is represented as a multinomial distribution of degree n, $B(C) = \langle p_{Aberdeen}=0.21, p_{Abileen}=0.18, p_{Albany}=0.13, p_{Albuquerque}=0.10, \ldots p_{Yuma}=0.02 \rangle$. The corresponding **k+other** belief representation only keeps track of the most likely, or top k hypotheses. We denote the hypothesis with the highest probability score h1, the one with the second highest probability h2, and the one with the third highest probability h3. The event space for this compressed belief therefore is: {h1, h2, h3, other}. Note that this representation abstracts away from the actual concept values; the system keeps track of the actual values that correspond to h1, h2, and h3 separately. The compressed belief E(C) consists therefore of a multinomial distribution of degree k+1: $e = \langle p_{h1}=0.21, p_{h2}=0.18, p_{h3}=0.13, p_{other}=0.48 \rangle$, and a top-k hypotheses mapping S = {h1=Aberdeen; h2=Albany; h3=Abileen} that maintains the identities of the top-k most likely hypotheses.

The proposed **k+other** belief compression is lossy. We only keep track of probabilities for k most likely hypotheses at a time. For instance, in the example illustrated in Figure 74, we have lost information about the individual probabilities for Albuqueque, Allentown, … Yuma. The corresponding probability mass is clustered under the "other" category. This does not pose a significant problem if k is chosen sufficiently large. As we have argued before, the number of alternate hypotheses a system will hear throughout a conversation is relatively small, and will rarely exceed k. On the

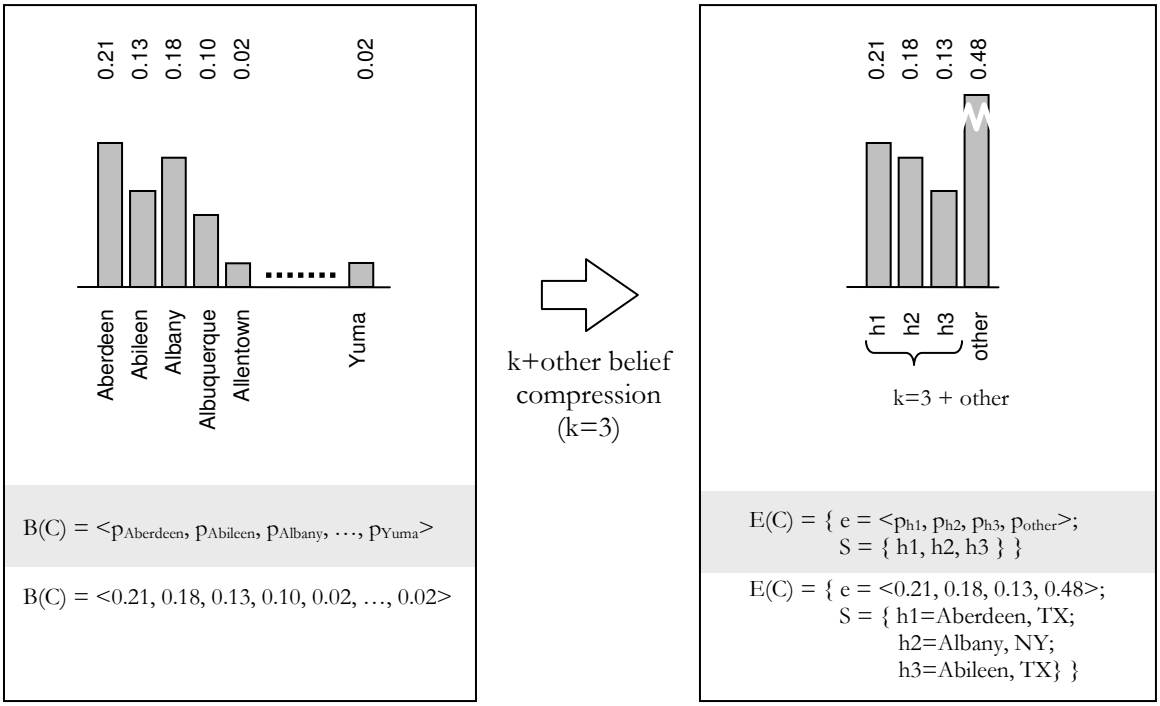


Figure 74. "k+other" belief compression

other hand, this representation leads to a significant reduction in the number of parameters needed to model the belief: from n, the cardinality of the concept (which we have seen can be quite large), down to k – a small integer. Using this representation, a supervised learning approach to the belief updating problem becomes tractable. The target variable is a multinomial of small degree.

Secondly, the proposed **k+other** belief representation abstracts away from the actual concept values, and therefore allows us to construct a concept-independent belief updating model. The model performs updates in an abstracted belief space: {h1, h2, h3, … , hk, other}. The values corresponding to the top k hypotheses are stored separately. The cardinality of the concept becomes irrelevant, since all concepts are reduced to a **k+other** representation. Using this representation, we can construct a model that answers questions such as: if the system initially believed that the probability for the top hypothesis h1 was p_{h1}=0.21, how should the probability for this hypothesis be updated in the next time step given a certain system action and user response? Note that the computation is not specific to the actual value for h1 (Aberdeen). Nevertheless, abstracted information about this value, such as "does h1 appear in the user response again?", or "what is the prior likelihood of h1?" can be used in the computation since the system has access to the actual value at runtime through the S mapping.

Finally, "empty concept" beliefs (i.e. the concept has not yet been specified by the user) are naturally encoded in the **k+other** representation by assigning the full probability mass to the "other" category: {e=<p_{h1}=0, p_{h2}=0, … p_{hk}=0, p_{other}=1>; S={h1=Ø, h2=Ø, … hk=Ø}}.

6.4.1.2 Dynamic belief space

Using the proposed **k+other** belief representation, the belief updating problem becomes:

$$E_{t+1}(C):\{e_{t+1}, S_{t+1}\} \leftarrow f\ (\ E_t(C):\{e_t, S_t\}, SA_t(C), R_t\ )$$

The task of generating an updated compressed belief e_{t+1} can be cast as a supervised learning problem: given a set of features characterizing the initial belief $E_t(C)$, system action $SA_t(C)$, and user response $R_t(C)$, compute the multinomial variable e_{t+1}. One issue remains. Throughout the interaction, the model needs to update not only the variable e, but also the top-k hypotheses mapping S. In

other words, we would like to be able to dynamically add and drop hypotheses from the abstracted belief space. Initially the top-k hypotheses mapping is S_0={h1=Ø, h2=Ø, … hk=Ø}. As the system starts hearing hypotheses for this concept, this mapping should be updated accordingly. Furthermore, if more than k hypotheses are heard throughout the conversation, some of the low-probability old hypotheses should be dropped in order to make space for the newer hypotheses.

This problem is addressed as follows. Let m and n be two small integers, such that k=m+n. In each step, the new abstracted belief space is constructed by retaining the top-m of the initial k hypotheses, and adding the top-n new hypotheses from the user response. Formally, let S_t={h1, h2, … h3, other}. We assume that, in the most general case, the user response R_t can contain more than one hypothesis for the concept C (for instance, if R_t is an n-best list). Let RH_t={r1, r2, …, rp} be the set of new hypotheses for concept C that are present in the user response R_t. Note that these are **new** hypotheses, i.e. they are not already present in S_t. ($RH_t \cap S_t = \emptyset$). Then, the new abstract belief space at time t+1 is S_{t+1} = {h1, h2, …hm, r1, r2, rn, other}. Consequently, e_{t+1}= <p_{h1}, p_{h2}, … p_{hk}, p_{r1}, p_{r2}, … p_{rn}, p_{other}>.

For clarity, we show a concrete, detailed example of how the proposed belief space and probabilities are updated throughout a conversation in Figure 75. The example illustrates three consecutive belief updating steps for the `departure_city` concept in a spoken dialog system that handles flight reservations. The belief updating model illustrated in this example is a **4[2;2]+other** model, i.e. it tracks up to k=4 simultaneous alternate hypotheses for the concept, and it each step it keeps the top m=2 initial hypotheses and adds up to n=2 new concept hypotheses from the recognition n-best list. Note that the system in the given example uses a 3-best list returned by the recognition engine.

As illustrated in Figure 75, the proposed approach allows us to dynamically add and drop hypotheses from the belief space, while maintaining the abstraction over the actual concept values. The belief updating model can track up to k different values for a concept simultaneously, and integrates information across multiple turns in a conversation. The model can also integrate information from multiple recognition hypotheses (up to n alternative values in each time-step). Note that, in the degenerate case when m=0, the model **k[m=0;n=k]+other** essentially performs an n-best list rescoring task.

Using the proposed **k[m;n]+other** abstraction, the belief updating problem reduces to:

$$e_{t+1}(C) \leftarrow f\ (\ E_t(C):\{e_t, S_t\}, SA_t(C), R_t\ )$$

where $e_{t+1}(C)$ = <p_{h1}, p_{h2}, … p_{hk}, p_{r1}, p_{r2}, … p_{rn}, p_{other}> is a multinomial distribution of degree k+1 that captures the probabilities that the correct concept values is one of the initial top-m values (h1, h2, … hm) or one of the new top-n values from the recognition result (r1, r2, … rn), or something else (p_{other})

6.4.1.3 Belief updating as a multinomial regression task

Using the proposed compressed belief representation, the 1-step belief updating problem can be stated as a multinomial regression task: given a set of features that characterize the initial system belief at time t, $E_t(C):\{e_t, S_t\}$, the system action $SA_t(C)$ and the corresponding user response R_t, compute the updated belief (multinomial distribution) $e_{t+1}(C)$.

$$e_{t+1}(C) \leftarrow f\ (\ Et(C):\{e_t(C), S_t(C)\}, SA_t(C), R_t\ )$$

This problem can be addressed by constructing a multinomial logit model, also known as a multinomial generalized linear model [76]. The multinomial logit model is a natural extension of the logistic regression model, for the case when the response variable is multinomial. If $\overline{F}$ is a set of features (including a constant feature for the intercept), and $\overline{Y}$ = <y_1, y_2, … y_N> is a multinomial response variable of degree N, then the multinomial logit model $\overline{Y} \leftarrow \overline{F}$ has the form:

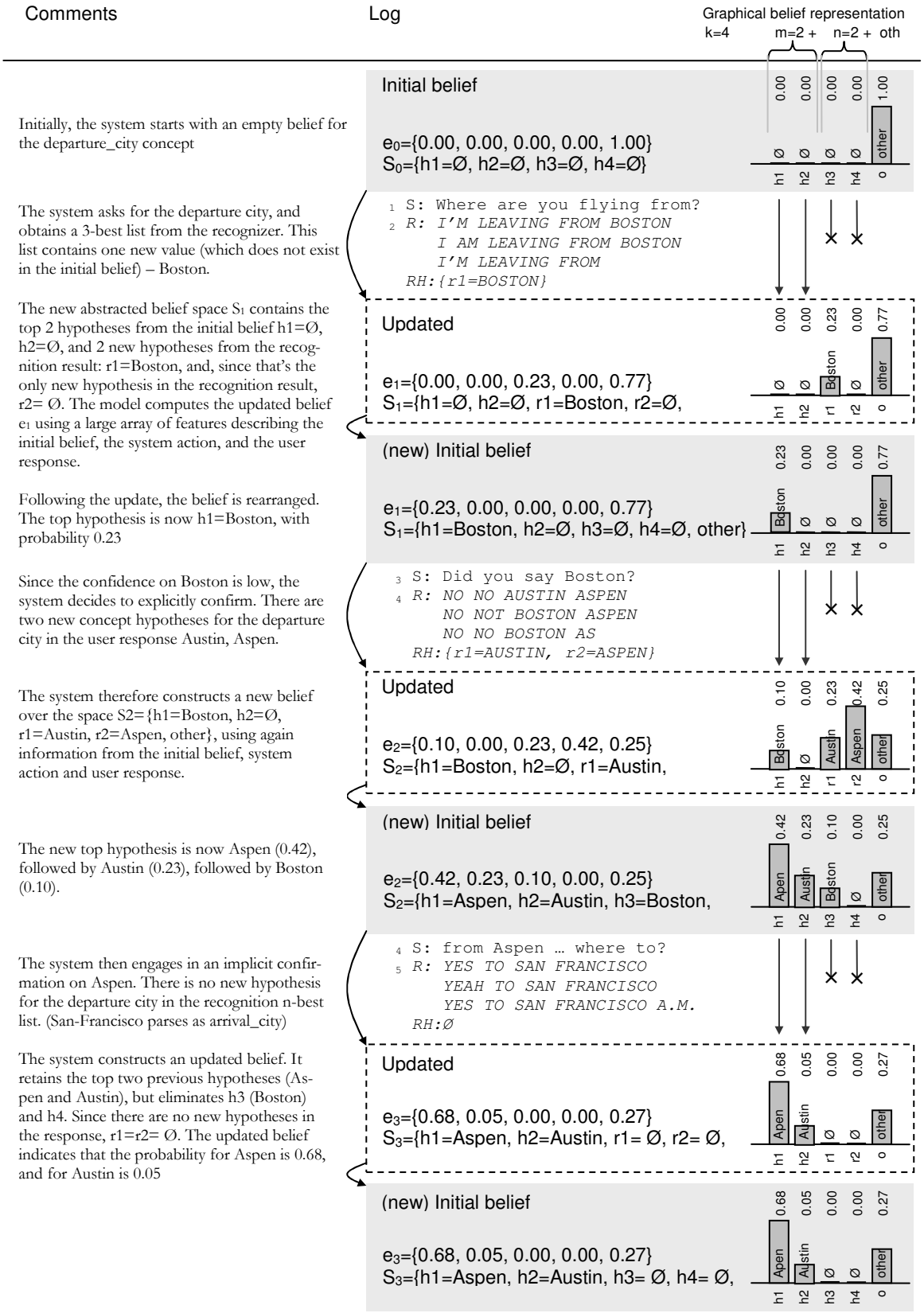


Figure 75. Three consecutive belief updating steps with a **4[2;2]+other** belief updating model

$$P(\overline{Y} = y_i \mid \overline{F}) = \frac{e^{\overline{\alpha_i} \cdot \overline{F}}}{1 + \sum_{j=1}^{N-1} e^{\overline{\alpha_j} \cdot \overline{F}}}, i = \overline{1, N-1},$$

$$P(\overline{Y} = y_N \mid \overline{F}) = \frac{1}{1 + \sum_{j=1}^{N-1} e^{\overline{\alpha_j} \cdot \overline{F}}}$$

In the belief updating problem, the response variable is $Y=e_{t+1}(C)=<p_{h1}, p_{h2}, \ldots, p_{hm}, p_{r1}, p_{r2}, \ldots p_{rn}, p_{other}>$ of degree k+1, and the set of features $\overline{F}$ characterizes the initial belief over the concept $E_t(C)$, the system action $SA_t(C)$ and the user response R_t. The multinomial logit model in this case computes:

$$P(C = S_t\{hi\} \mid \overline{F}) = \frac{e^{\overline{\alpha_{hi}} \cdot \overline{F}}}{1 + \sum_{j=1}^{m-1} e^{\overline{\alpha_{hj}} \cdot \overline{F}} + \sum_{j=1}^{n-1} e^{\overline{\alpha_{rj}} \cdot \overline{F}}}, i = \overline{1, m-1}$$

$$P(C = S_t\{ri\} \mid \overline{F}) = \frac{e^{\overline{\alpha_{ri}} \cdot \overline{F}}}{1 + \sum_{j=1}^{m-1} e^{\overline{\alpha_{hj}} \cdot \overline{F}} + \sum_{j=1}^{n-1} e^{\overline{\alpha_{rj}} \cdot \overline{F}}}, i = \overline{1, n-1}$$

$$P(C = \text{other} \mid \overline{F}) = \frac{1}{1 + \sum_{j=1}^{m-1} e^{\overline{\alpha_{hj}} \cdot \overline{F}} + \sum_{j=1}^{n-1} e^{\overline{\alpha_{rj}} \cdot \overline{F}}}$$

The model parameters can be estimated in a maximum likelihood fashion from training data. The total number of parameters in this model is $k \cdot \#F$, where #F is the number of features used in the model. For small values of k, this model can be trained using relatively little data (more details about the scalability of the proposed approach will be presented later). Similar to logistic regression models, multinomial logit models can be constructed in a step-wise fashion: a large number of features are inspected in each step, and the feature that leads to the best improvement is added. The Bayesian Information Criterion can be used to decide when to stop adding features to the model. This approach allows us to investigate and leverage a very large number of potential features, without having explicit knowledge of the relationships between these features.

6.4.2 Data

We now describe the data that was used to train the proposed 1-step belief updating models.

6.4.2.1 System

The training data was collected using RoomLine, a telephone-based, mixed-initiative spoken dialog system. The system, described in more detail in 3.4.1 from Chapter 3, assists users in performing conference room reservations in two buildings on the CMU campus. The system knows about the schedules and characteristics (e.g. size, location, a/v equipment) of 13 conference rooms, and can engage in a negotiation dialog to identify the room that best matches the user's needs.

The RoomLine dialog manager operates with a set of 28 concepts of different cardinalities. 10 of these are trigger concepts, which can take only one possible value except for "empty"; these concepts are used to capture user commands, like "help!", "repeat!", etc. In addition, RoomLine uses 8 Boolean concepts that can take 2 possible values (yes / no). Finally, there are 10 non-Boolean concepts with cardinalities ranging from 1 to 500. Table 22 presents a brief summary.

Concept type (name)	Example possible values	Cardinality
10 trigger concepts	trigger concepts have only one possible value, i.e. "on", besides being empty.	1
8 boolean concepts want_reservation_single, want_summary, satisfied, anything_else, etc.	yes/no	2
10 non-boolean concepts:		
start_time	9:15, 8:30, etc (every 15 minutes during the day)	96
end_time	9:15, 8:30, etc (every 15 minutes during the day)	96
date	any date in the year	365
room_equip	projector, network, whiteboard, computer	4
room_size	15, 200, etc (# of people the room should hold)	500
room_size_spec	small, large, smaller, larger, smallest, largest	6
room_location	wean hall, newell simon hall	2
room_number	7220, 8220, 4625, etc (13 room numbers)	13
user_name	"guest user"	1
timeperiod_spec	morning, afternoon	2

Table 22. Concepts in the RoomLine system

6.4.2.2 Data collection experiment and corpus

The data used for training and evaluating the belief updating models was collected through a user study, described later in Chapter 8, subsection 8.3.1. Here, we briefly present a number of details and statistics pertinent to the belief updating work.

The corpus contains 449 sessions and 8278 user turns. The user turns were orthographically transcribed and checked by a second annotator. Based on these transcriptions, we manually annotated whether or not the system misunderstood the user (e.g. turn-level misunderstanding), and whether or not the user was trying to correct the system (e.g. turn-level correction). Additionally, we also annotated the correct concept values, as they were specified by the user, at each point in the dialog. This latter information was used to generate the target labels for the multinomial regression belief updating models.

6.4.2.3 System actions

The RoomLine system engages in 5 types of actions with respect to any given concept. The actions are illustrated in Table 23.

The first example shows a **request** action on the `start_time` concept. This happens when

request	S: For what time do you need the room? *U: 10 a.m.* *R: TEN A_M / 0.35*
explicit confirmation	S: Did you say 10 a.m.? *U: yes until noon* *R: GUEST UNTIL ONE / 0.87*
implicit confirmation	S: starting at 2p.m. ... until what time? *U: I need a different time* *R: CAN YOU DETAILS TIME / NONU-0.00*
unplanned implicit confirmation	S: I found 3 rooms Monday between 2 and 4 p.m. Would you like a small one or a large one? *U: no not Monday, Friday* *R: ABOUT MONDAY THURSDAY / 0.22*
no action	S: Okay. To make the reservation, please tell me your name, or say 'guest user' if you are not registered with the system *U: guest user* *R: NEXT TUESDAY / 0.62*

Table 23. System actions in the RoomLine corpus

the system explicitly asks for the value for a concept.

The second example shows an **explicit confirmation** action on the same `start_time` concept. The system explicitly asks a simple yes-no question to verify the current value of the `start_time` concept.

The third example shows an **implicit confirmation** action on the `start_time` action. The system echoes back the current value, and continues with the next question. Note that in a single turn the system might engage in multiple actions with respect to multiple concepts. In the example fromTable 23, the system coupled the implicit confirmation on the `start_time` concept with a request for the `end_time` concept. The belief updating mechanism works independently on these two concepts; each concept is updated separately and the system action with respect to the concept to be updated is taken into account. In this case, this single turn generates two learning instances for the belief updating model: one for the `start_time` concept, and one for the `end_time` concept.

The fourth example shows **unplanned implicit confirmation** actions. In this case, the system prompt includes information previously acquired from the user (e.g. the `date`, the `start_time` and the `end_time`). This information is echoed back to the user, but this is **not** part of a planned implicit confirmation action like in the third example. Rather, this happens as a side-effect of the system prompt design. The system prompt from the fourth example therefore includes three unplanned implicit confirmations (one for the date, one for the `start_time` and one for the `end_time`), as well as one request action for the `room_size` concept. This turn will generate four learning instances for the belief updating model.

Finally, the last example illustrates a special case: **no-action**. In this example, the system does not engage in any request or confirmation action with respect to the date concept. Nevertheless the system hears a value for this concept – *"NEXT TUESDAY"*. In this case, the value for the date concept appears as a result of misrecognition. However, in a mixed-initiative spoken language interface, this can also happen because the user can take the initiative or provide information that was not directly requested by the system. Furthermore, in many cases users do not correct the system immediately after a mistake (see more in section 6.4.3). As a consequence, we argue that spoken language interfaces should update their beliefs not only for the concepts requested or confirmed in the previous turn, but also for all concepts heard in the user response. In fact, in section 6.6, we discuss ideas for extending this further to all concepts the system operates with.

Note that the five action types we have described above are domain-independent. At the same time, the belief updating approach described in this chapter is not tied to any particular action set. For instance, if a spoken dialog system also uses a disambiguation action (e.g. "Did you say Austin or Aspen?"), this could be easily added to the set described above.

6.4.2.4 Belief updating heuristics

Throughout the data collection experiment, the system kept track of multiple alternate hypotheses for each concept, together with their confidence scores. The system used both explicit and implicit confirmations to guard against potential misunderstandings. These strategies were engaged following a commonly used confidence threshold model, combined with an epsilon-exploration approach. With probability epsilon (0.2 in our case), the system would take a random action from the set {accept, explicit confirm, implicit confirm}. With probability 1-epsilon, the system would use 2 thresholds to decide the action: if the confidence score for the top hypothesis was above 0.8 the system would accept that value; if the confidence score was between 0.5 and 0.8 the system would implicitly confirm the value; otherwise, the system would explicitly confirm the value. The system also used a rejection threshold of 0.3. Finally, note that the implicit confirmation action was not available on all concepts. For instance, the system never engaged in implicit confirmations on trigger concepts.

RoomLine used a set of simple heuristic rules to perform belief updating. After explicit confirmations the system boosted the confidence score to 0.95 when the perceived user response contained a positive marker (e.g. yes, right, etc.) and deleted the hypothesis if the response included a negative marker (e.g. no, wrong, incorrect, etc.) For implicit confirmations, the system would dis-

credit the hypothesis if it heard a negative marker in the user response and the implicit confirmation was not followed by a yes/no question; if the user response included a new value for the confirmed concept, the system overwrote the old value (hence deleted the initial hypothesis); finally, for all other responses to implicit confirmations the system assumed that the initial hypothesis was correct, and increased its confidence score to 0.95. In general (as opposed to after a confirmation action), the system used a naïve probabilistic update rule to update its beliefs. It multiplied the initial probability distribution (belief over the concept) with the new distribution coming from the recognizer, and renormalized. Note that the naïve probabilistic update rule was applied during unplanned implicit confirmations (see section 6.4.1.2), because the dialog manager did not plan and was not direclty aware of the confirmation action.

6.4.3 An empirical investigation of user responses to confirmation actions

To gain a better understanding of the challenges that we are facing with respect to belief updating, we performed an analysis of user responses to system confirmation actions.

A similar analysis was previously conducted by Krahmer et al [63]. The authors analyzed user responses to explicit and implicit confirmations in a corpus of 120 dialogs from a Dutch train-table information system. Several positive and negative cues for user corrections were identified and a distributional analysis revealed that some of these cues have good predictive capacity to signal errors. Furthermore, the study confirmed that user responses to system confirmation actions can be fairly complex. Even for explicit confirmations, which are simple yes/no questions, user responses do not always incorporate confirmation or disconfirmation markers (e.g. yes, no, that's right, wrong, etc.)

Because we were interested in comparing results across domains, we followed the same methodology as Krahmer et al [63]. We divided the user responses into the 3 response types: `yes`, `no` and `other`. This classification was performed twice: once using the transcripts, and once using the recognition hypotheses. Next, we cross-tabulated these classes against whether the system was atempting to confirm a correct or an incorrect concept value. The results are presented in Table 24.A-F.

As Table 24.A shows, user responses following explicit confirmations included positive markers in 94% of the cases when the valued to be confirmed was indeed correct. In contrast, the answers included a negative marker in only 72% of the explicit confirmations with incorrect values. This discrepancy is consistent with prior observations made by Krahmer et al [63] (the numbers from their study are shown in brackets in the Table 1.A). In a significant number of cases (27%) users attempted to correct the system by other means than a negative answer to the explicit confirmation question. A closer look at these instances reveals that in most of these cases (~70%) users repeated

A. Explicit Confirmation (from transcripts)

	Yes	No	Other
Correct (1159)	**94%** [93 %]	**0%** [0%]	**5%** [7%]
Incorrect (279)	**1%** [6%]	**72%** [57%]	**27%** [37%]

B. Explicit Confirmation (from decoded results)

	Yes	No	Other
Correct (1159)	**87%**	**1%**	**12%**
Incorrect (279)	**1%**	**61%**	**38%**

C. Implicit Confirmation (from transcripts)

	Yes	No	Other
Correct (554)	**30%** [0%]	**7%** [0%]	**63%** [100%]
Incorrect (229)	**6%** [0%]	**33%** [15%]	**61%** [85%]

D. Implicit Confirmation (from decoded results)

	Yes	No	Other
Correct (554)	**28%**	**5%**	**67%**
Incorrect (229)	**7%**	**27%**	**66%**

E. Unplanned Implicit Conf. (transcripts)

	Yes	No	Other
Correct (2725)	**25%**	**19%**	**56%**
Incorrect (457)	**12%**	**28%**	**60%**

F. Unplanned Implicit Conf. (decoded)

	Yes	No	Other
Correct (2725)	**24%**	**17%**	**59%**
Incorrect (457)	**10%**	**21%**	**69%**

Table 24. Cross-tabulation of response-type against correct and incorrect system confirmation actions. Percentages reflect proportions from the correct and respectively incorrect confirmations. Bracketed numbers were reported by Krahmer and Swerts in a similar study of a Dutch train-timetable information system [63]

the correct value for the concept. Other times, users ignored the system's explicit confirmation and tried to change the focus of the conversation. Prompted by this observation, we cross-tabulated users' correction attempts versus the correctness of the value the system tried to confirm. The results are shown in Table 25.A. We noted that 10% of the incorrect explicit confirmations (29 of the 29+250=279 cases) were not corrected by the users – these are mostly the cases in which users were trying to shift the focus of the conversation. Finally there were a small number of `other`-type responses on explicit confirmations that stemmed from audio end-pointing errors and turn-overtaking issues.

A comparison between Table 24.A and Table 24.B reveals an increased proportion of the `other`-type responses in the decoded results. This difference is explained by misrecognitions of yes / no into acoustically similar words (e.g. this / small). The difference is also present for both planned and unplanned implicit confirmations, and highlights again that the presence of recognition errors increases the difficulties of detecting corrections and building accurate beliefs.

For implicit confirmations, there are much fewer `yes`- and `no`-type responses, and the `other`-type responses dominate (see Table 24.C). For correct implicit confirmations this large number is expected: users mostly answer the follow-up system question. The large number of `other`-type responses on incorrect implicit confirmations is explained by the fact that often users will avoid correcting the system in this situation. Table 25.B. shows that for 51.5% (118 / 118+111=229) of incorrect implicit confirmations users are not attempting to correct the system.

A closer analysis of these instances led to several observations. First, although in these 118 cases users did not immediately correct the system's incorrect implicit confirmation, in 49 of these cases (~40%) users did attempt to correct the error in a later turn. More interestingly, the analysis revealed that users interact with the system strategically: they correct the errors which **have to be recovered** for the successful completion of the task (we shall refer to these as **critical errors**), and mostly ignore the others. Given the nature of the domain, certain scenarios could be completed by users in different ways. This in turn rendered some concepts more important for task success than others. For instance, if the system accidentally misrecognized that the user wanted a room with a whiteboard, this incorrectly filled concept (equipment) had no impact on task success if the goal was only to make a reservation for a room on Friday morning from 10 to 11. A user could ignore this system error altogether and still complete the task successfully. Out of the 49 later-turn corrections, 47 were on critical errors, and only 2 on non-critical errors. At the same time, there were only 14 critical errors that were never corrected, and hence had a direct contribution to task failure. These results confirm that users adapt their behaviors to their specific goals, and engage in corrections only when it is necessary to do so.

Finally, for unplanned implicit confirmations, the distribution of response types is similar to the one for the planned implicit confirmations, and the same phenomenon of users not always correcting the system can be observed.

Our analysis in the RoomLine domain corroborates previous observations made by Krahmer et al [63]. The results indicate that user responses to system confirmation actions span a wide language spectrum. In consequence, simple heuristics for updating beliefs (e.g. looking for confirmation and disconfirmation markers in the user responses) will fall short both in terms of accuracy and coverage. In the next three subsections, we discuss several supervised-learning based belief updating models that leverage a large set of user response features in order to construct more accurate beliefs.

A. Explicit Confirmation		
	User corrects	User does not correct
Correct	0	1159
Incorrect	250	29

B. Implicit Confirmation		
	User corrects	User does not correct
Correct	2	552
Incorrect	111	118

C. Unplanned Implicit Confirmation		
	User correct	User does not correct
Correct	11	2714
Incorrect	138	319

Table 25. Users' correction attempts versus correctness of confirmed values.

6.4.4 The $BU^{m=1}_{n=0}$ model: updating beliefs after confirmation actions

6.4.4.1 Model

We begin with the simplest belief updating model: $BU^{m=1}_{n=0}$ (k=1, m=1, n=0). This model updates the confidence score for the initial top hypothesis in light of the user response to a system action. The belief space is static, and no new concept hypotheses are accumulated throughout the interaction (n=0). As a consequence, this model is useful only for the cases when the system already has an initial top hypothesis for a concept, and attempts to verify it. The model was constructed for the three confirmation actions: explicit confirmation (`EC`), implicit confirmation (`IC`) and unplanned implicit confirmation (`UIC`).

In the $BU^{m=1}_{n=0}$ model, the multinomial output variable e_{t+1} reduces to a simple binary variable $e_{t+1}=<p_{h1}, p_{other}>$. The multinomial regression model reduces to a logistic regression model that estimates the likelihood of the initial top hypothesis in light of the system action and user response:

$$P(C = S_t\{h1\} \mid \bar{F}) = p_{h1} = \frac{e^{\bar{\alpha}\cdot\bar{F}}}{1+e^{\bar{\alpha}\cdot\bar{F}}}$$

$$P(C = other \mid \bar{F}) = p_{other} = \frac{1}{1+e^{\bar{\alpha}\cdot\bar{F}}}$$

The learning instances for this model are individual concept updates. The target values for training the model, e_{t+1}, are provided by the manual labeling of correct concept values. If the correct concept value following an update is indeed h1, then $e_{t+1}=1$, otherwise $e_{t+1}=0$. During training the vector of model parameters $\bar{\alpha}$ is estimated in a maximum likelihood procedure.

6.4.4.2 Features

We characterized the initial system belief and the user response in terms of a large set of features $\bar{F}$, extracted from different knowledge sources in the system. The complete set of features is presented in Table 26. Note that this table also includes features which are not used in the BU^{1}_{0} model, but only in latter, more complex belief updating models.

The features can be classified broadly in the following categories:

- **features describing the initial system belief:** these include information about the identity of the concept undergoing the update, the confidence scores and availability of the initial hypotheses for this concept, as well as information on whether the initial top hypothesis had already been explicitly confirmed or disconfirmed at the time of the update.
- **features describing new concept values in the user response:** these include information about the presence of new concept hypotheses in the user response, as well as the associated confidence scores. (For the purposes of the $BU^{m=1}_{n=0}$ belief updating model, the user response consists of the selected recognition hypothesis; for some of the more advanced models discussed later, this is extended to include multiple alternate recognition hypotheses).
- **features describing the prior likelihood of the initial and new concept hypotheses:** a domain expert, who did not have access to the training and evaluation corpora manually constructed priors for three of the most important concepts in the system: `start_time`, `end_time` and `date`. For all other concepts, we assumed uniform priors.

Table 26. Features for belief updating

The feature types are encoded as follows: R=real, C=count, N=nominal, B=boolean
The derived features are encoded as follows:
>m = binary version indicating if the feature value is greater than the mean value of the feature in the dataset
>0 = binary version indicating if the feature value is greater than 0
>1 = binary version indicating if the feature value is greater than 0
>2 = binary version indicating if the feature value is greater than 0
>4 = binary version indicating if the feature value is greater than 0
dtf = difference between the current feature value and the feature value in the first turn in the dialog
dtp = difference between the current feature value and the feature value in the previous turn in the dialog

We use the term *concept hypothesis* to denote a possible value for a concept (e.g. 2a.m. is a concept hypothesis for the start_time concept); we use the term *recognition hypothesis* to denote a hypothesis generated by the speech recognizer. In our setup, two different recognizers were used (one using acoustic models trained with data from male speakers, one using models trained with data from female speakers). After the confidence score is assigned only the recognition hypothesis with the highest confidence score is forwarded to the dialog manager. We use the term *selected recognition hypothesis* to denote the hypothesis that was forwarded to the dialog manager.

Feature name	Type	Derived features	Feature Description
Features characterizing the initial belief (before the update)			
concept_id	B		set of binary features capturing the identity of the concept undergoing the update (there RoomLine system operates with a set of 38 different concepts)
i_h1_confidence	C	>m	the confidence score for the initial (before the update) top concept hypothesis
i_h2_confidence	C	>m	the confidence score for the initial second concept hypothesis
i_h3_confidence	C	>m	the confidence score for the initial third concept hypothesis
i_h1_avail	B		indicates that a top concept hypothesis exists in the initial belief
i_h2_avail	B		indicates that a second concept hypothesis exists in the initial belief
i_h3_avail	B		indicates that a third concept hypothesis exists in the initial belief
i_h1_explicitly_confirmed_already	B		indicates that the initial top concept hypothesis has already been explicitly confirmed by the user (prior to this update)
h_h1_explicitly_confirmed_already	B		indicates that the history value for this concept has already been explicitly confirmed by the user (prior to this update)
iorh_h1_explicitly_confirmed_already	B		indicates that either the initial top concept hypothesis or the history value (if the concept was recently reopened and no top concept hypothesis exists) has been explicitly confirmed by the user (prior to this update)
Features characterizing new values in the user response (the selected recognition hypothesis)			
srh_h_h1_avail	B		the history value for the concept is again present in the selected recognition hypothesis
srh_i_h1_avail	B		the initial top concept hypothesis is present in the selected recognition hypothesis
srh_i_h1_confidence	C		the confidence score for the top concept hypothesis in the selected recognition hypothesis
srh_iorh_h1_avail	B		the initial top concept hypothesis or the history value for the concept is again present in the selected recognition hypothesis

srh_i_h2_avail	B		the initial second concept hypothesis is again present in the selected recognition hypothesis
srh_i_h2_confidence	C		the confidence score for the initial second concept hypothesis in the selected recognition hypothesis
srh_i_h3_avail	B		the initial third concept hypothesis is again present in the selected recognition hypothesis
srh_i_h3_confidence	C		the confidence score for the initial third concept hypothesis in the selected recognition hypothesis
srh_r1_avail	B		the selected recognition hypothesis contains a new value for the concept, which is different from the values in the initial belief (r1)
srh_r1_confidence	C	>0.25, >0.5, >0.75	the confidence score for the new concept hypothesis in the selected recognition hypothesis
srh_r1_explicitly_disconfirmed_already	B		the new concept hypothesis present in the selected recognition hypothesis was already disconfirmed earlier
Priors			
i_h1_prior	C	>1	prior score for the initial top concept hypothesis
i_h2_prior	C	>1	prior score for the initial second concept hypothesis
i_h3_prior	C	>1	prior score for the initial third concept hypothesis
h_h1_prior	C	>1	prior score for the history value for the concept
iorh_h1_prior	C	>1	prior score for the initial top concept hypothesis or for the history value (whichever is available)
srh_r1_prior	C	>1	prior score for the new concept hypothesis contained in the selected recognition hypothesis
Confusability scores			
i_h1_confusability	C	>m	confusability score for the initial top concept hypothesis
i_h2_confusability	C	>m	confusability score for the initial second concept hypothesis
i_h3_confusability	C	>m	confusability score for the initial third concept hypothesis
h_h1_confusability	C	>m	confusability score for the history value for the concept
iorh_h1_confusability	C	>m	confusability score for the initial top concept hypothesis or for the history value for the concept (whichever is available)
srh_r1_confusability	C	>m	confusability score for the new concept hypothesis contained in the selected recognition hypothesis
Other features characterizing the user response: speech recognition features			
am_score	R		the acoustic model score
lm_score	R		the language model score
decoder_score	R		the decoder score
frame_num	C	>m	the number of frames
word_num	C	>1, >2, >4	the number of words
word_num_class	N		nominal feature indicating whether the selected recognized hypothesis contains 1 word, 2 words, 3 or 4 words, or more than 4 words
unconf_num	C		number of unconfident words (Sphinx tags individual words as unconfident if no trigram is found in the language model ending in the current word, and a bigram back-off is forced)
Other features characterizing the user response: prosody features			
pitch_mean	R	>m, dtf, dtp	the pitch mean
pitch_range	R	gtm, dtf, dtp	the range of the pitch (max – min)
pitch_min	R	>m, dtf, dtp	the minimum pitch level

pitch_max	R	>m, dtf, dtp	the maximum pitch level
pitch_std	R	>m	the pitch standard deviation
pitch_min_slope	R		the minimum value for the pitch slope (the largest pitch decrease)
pitch_max_slope	R		the maximum value for the pitch slope (the largest pitch increase)
pitch_max_increase	R		the maximum pitch increase
pitch_max_decrease	R		the maximum pitch decrease
num_voiced_segments	C		the number of voiced segments
rate_voiced_segments	C		the ratio of voiced segments (out of the total number of voiced and unvoiced segments)
perc_unvoiced	R	>m	the percentage of the selected recognized hypothesis (in frames) that is not voiced
prepau	R	>m	the length of the initial unvoiced segment (initial pause)
Other features characterizing the user response: lexical features			
mark_confirm	B		presence of confirmation markers
mark_disconfirm	B		presence of disconfirmation markers
answer_type	N		3-class answer type: yes-type answer, no-type answer or other-type answer
lex	B		a set of binary features that captures the presence of 10 words correlated with misunderstandings.
lexw1	B		same as above, but with the condition that the selected recognition hypothesis contains only one word
Other features characterizing the user response: language understanding features			
no_parse	B		indicates that no parse was constructed for the selected recognition hypothesis
slot_num	C		number of grammar slots
rep_slots_num	C	>0	number of repeated grammar slots (wrt the previous turn)
new_slots_num	C	>0	number of new grammar slots (wrt the previous turn)
uncov_num	C	>0	number of words not covered by the parse
frag_num	C		the number of fragments in the parse
gap_num	C	>0, >1	the number of gaps in the parse
hyp_num_parses	C		the number of alternative parses generated for the selected recognition hypothesis (due to grammar ambiguities, Phoenix can sometimes generate multiple parses for each recognition hypothesis)
total_num_parses	C		the total number of alternative parses generated for this user input (e.g. considering the recognition hypotheses from both male and female decoders)
Other features characterizing the user response: inter-recognition hypotheses features			
ih_diff_lexical	B		the two recognition hypotheses from the male and female recognition engine are different
ih_diff_lexical_one_word	B		the two recognition hypotheses from the male and female recognition engine are different, and they are both one word long
Other features characterizing the user response: dialog-level features			
nonu	B		indicates that the current turn was a non-understanding
confidence	R	>0.25, >0.5, >0.75	the confidence score for the selected recognition hypothesis
slots_matched	C	>1	the number of grammar slots in the selected recognition hypothesis that matched an open expectation
slots_blocked	C	>0	the number of grammar slots in the selected recognition hypothesis that matched a closed expectation

first_level_matched	C	>0, >1	the first level in the expectation agenda where a slot from the current selected recognition hypothesis matched an open expectation
matched_in_focus	B		the selected recognition hypothesis matched the dialog expectation in focus (first level in the agenda)
barge_in	B		the user barge-in on the system
timeout	B		the user did not respond in time, and a time-out was triggered
Dialog state and history features			
turn	C	>0, >1, >m	the turn number
dialog_state_4	N		indicates whether the current dialog state is an open request (e.g. How may I help you?), a request for a Boolean concept (e.g. Would you like this room?) or a request for a non-boolean concept (e.f. For what time would you like this room?)
dialog_state_5	N		Indicates whether the current dialog state is an open request (e.g. How may I help you?), a request for a specific concept (e.g. For what time would you like this room?) or an explicit confirmation (e.g. Did you say you wanted the room for 10 a.m.?)
dialog_state_id	B		set of binary features capturing the state the dialog manager is in (there are 71 different states were encountered in the RoomLine corpus)
last_turn_nonu	B		indicates if the previous turn was a non-understanding
prev_confidence	C	>0.25, >0.5, >0.75	indicates the confidence score of the previous user turn
num_prev_nonu	C	>1, >2, >3	indicates how many consecutive non-understandings preceded the current user turn
num_prev_not_nonu	C		indicates how many consecutive turns that were not non-understandings preceded the current turn
h_avg_confidence	R	>m, >0.25, >0.5, >0.75	Indicates the mean confidence score up to this point in the dialog
h_ratio_nonu	R	>m	indicates the ratio of non-understandings up to this point in the dialog;

- **features describing the confusability of the initial and new concept hypotheses:** confusability for each concept value was computed empirically, from the training data. One confusability score was computed for each concept value encountered in the training set (for all other concept values, we assumed the mean confusability score). The score reflected the proportion of times that a concept value was recognized correctly; for instance: "out of all the times we had Tuesday in the recognition result, how many times did the user indeed say Tuesday?". Note that a higher value for this confusability score actually indicates that the concept is less likely to be confused with something else.
- **other features characterizing the user response:** these include speech recognition, prosodic, lexical, language understanding, as well as dialog level features characterizing the user response (for an in-depth description, see Table 26).
- **features characterizing the current dialog state and history:** these include information about the current state, the last turn (was it a non-understanding, what was the previous confidence score, etc), and the dialog history (e.g. the average ratio of non-understandings so far, and the average confidence score)

The constructed models did not use any features characterizing the system action $SA_t(C)$. Instead, separate models where constructed for each system action, because we expected that the user response features interact differently with the target variable depending on the system action. For instance, the presence of a positive marker like "yes" in the recognition result means something in the context of an explicit confirmation action, but something different in the context of a request action. We therefore separated the collected corpus of concept updates into 3 datasets, according to whether the system action was an explicit confirmation, an implicit confirmation, or an unplanned implicit confirmation, and we trained and evaluated a separate model for each dataset.

6.4.4.3 Empirical results

The 1-step belief updating models for each action were trained using a stepwise logistic regression approach. The next most informative feature (as measured by the improvement in average log-likelihood over the training data) was added to the model at each step. To prevent over-fitting, the Bayesian Information Criterion was used as a stopping mechanism.

We computed both the accuracy and the Brier score for the trained models in a 10-fold cross-validation process, and compared models' performance against three different baselines: **initial**, **heuristic** and **correction**.

The **initial** baseline reflects the accuracy (or error) of the initial system belief, at time t, before the system engages in an action and obtains the user response.

The **heuristic** baseline reflects the accuracy of the updated belief at time t+1, as it was constructed by the heuristic update rule currently used by the system. The heuristically updated beliefs should on average be more accurate than the initial beliefs; this is indeed the case, as the numbers we will present later show. At the beginning of this chapter, we have argued however that heuristic update rules face a number of difficulties. Our goal is to construct belief updating models that significantly improve upon these heuristic rules.

Finally, the **correction** baseline reflects the accuracy of a heuristic updating rule that has access to the actual transcript of the user response R_t. In other words, if we had perfect speech recognition, how well would we be able update our beliefs in one step? The heuristic rule used in conjunction with the transcripts essentially is: "assume the current top hypothesis is correct, unless the user corrects it, either by a disconfirmation marker, or by presenting a different value for the concept". It is important to notice that beliefs constructed based on this rule will not be 100% accurate, even if transcript information is used. One reason is that, as we have seen in our analysis from the previous section, users do not always correct the system in the turn following the confirmation. Secondly, even if users do correct the system, corrections cannot always be detected by simply looking for disconfirmation markers and new concept values (e.g. *"I need a different time"*). Nevertheless, the correction baseline provides a good target accuracy level for our belief updating models.

Action	Baselines			Models		
	initial	heuristic	correction	basic	full	runtime
EC (classif. error)	30.8%	16.5%	4.1%	4.9%	4.3%	4.4%
(Brier score)	0.1715	0.1201	0.0406	0.0375	0.0325	0.0345
IC (classif. error)	30.3%	26.1%	18.3%	16.4%	14.0%	14.0%
(Brier score)	0.2081	0.2180	0.1827	0.1277	0.1063	0.1063
UIC (classif. error)	15.2%	14.9%	12.4%	11.7%	9.4%	9.4%
(Brier score)	0.1267	0.1201	0.1239	0.0976	0.0760	0.0760

Table 27. Performance of $BU_{n=0}^{m=1}$ belief updating models (all models produce statistically significant improvements over the heuristic baseline; results in bold face represent statistically significant improvements over the correction baseline)

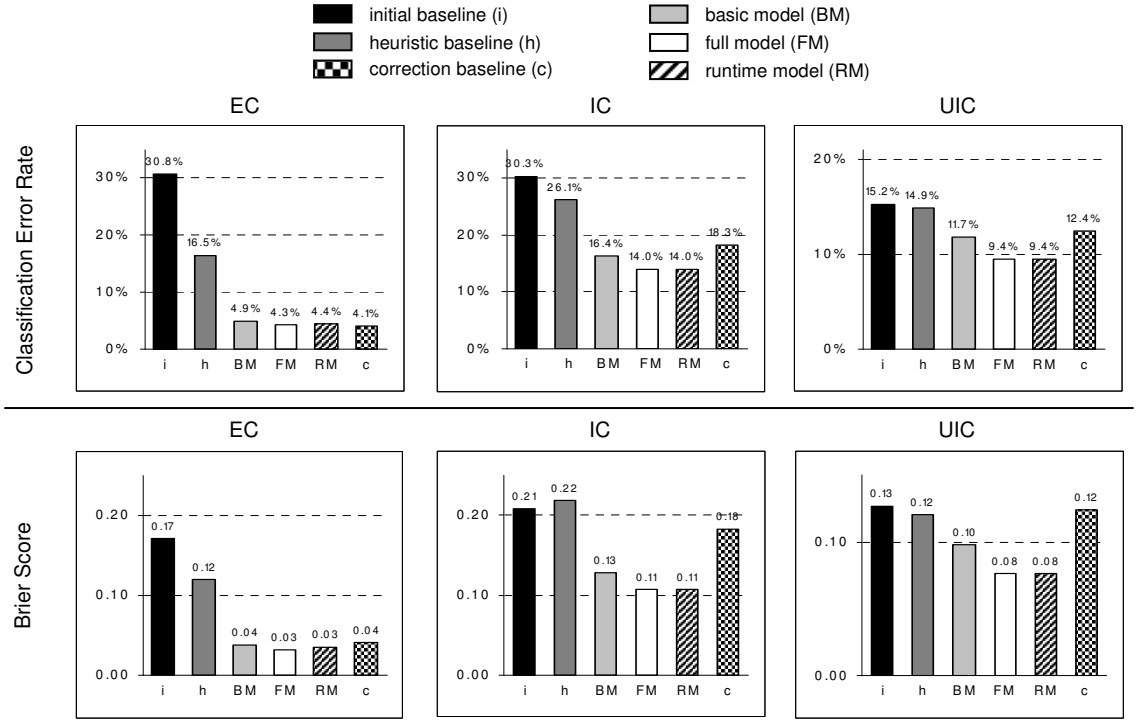


Figure 76. Performance of $BU_{n=0}^{m=1}$ belief updating models

For each system action we trained three different models, using three different feature sets: **basic**, **full** and **runtime**. The **basic model** (BM) uses all features, except for the priors and confusability scores. The **full model** (FM) uses the full set of features. Finally, the **runtime model** (RM) uses only the features available to the system at runtime, i.e. all the features except the prosody ones.

The results for the constructed models are presented in Table 27, and illustrated in Figure 76. For explicit confirmations, the initial baseline (i) error rate is 30.8% (in other words, in 30.8% of the cases when the system engaged in an explicit confirmation, the value that the system tried to confirm was incorrect). Applying the belief updating heuristic rule reduces this error to 16.4% (heuristic baseline – h). Finally, if we had access to the transcript and applied the heuristic belief updating rule, the error rate would decrease to 4.1% (correction baseline – c). All the proposed models construct beliefs that are significantly more accurate than the heuristic rule, and very close to the correction baseline: 4.9% using the basic features (basic model – BM), 4.3% if we add information about priors and confusability (full model – FM), and 4.4% if we use only the runtime features (runtime model – RM). The same effects can be observed in the Brier score evaluation (see Table 27, and the second row from Figure 76).

For implicit confirmations, the initial baseline is at 30.3%. The correction baseline is at 18.3%. Here, the relative improvement produced by the heuristic belief updating rule is smaller, from

30.3% to 26.1%. The heuristic rule closes only 35% of the gap between the initial and correction baseline (30.3-26.1 / 30.3-18.3). The weak performance of the belief updating heuristics in this case is somewhat expected, given our observations from the previous section. User responses following implicit confirmations cover a wider language spectrum (when compared to responses to explicit confirmations), and the belief updating task becomes more difficult. Nevertheless, the data-driven belief models performed significantly better, even exceeding the performance of the correction baseline: 16.4% error rate for the basic model, 14.0% for the full model and 14.0% for the runtime model. The data-driven models outperform the correction baseline by leveraging a number of contextual and pragmatic features that are not used by the correction baseline heuristic. For instance, information such as the identity of the concept undergoing the confirmation can play an important role because certain concepts are more often misunderstood than others. Additional information about barge-in, turn number, dialog state, concept priors and confusability, etc. also helps construct more accurate beliefs in a single time-step.

Finally, updating beliefs following unplanned implicit confirmations seems to be the most difficult of the three tasks. The heuristic baseline – 14.9%, produces only a minimal improvement over the initial baseline – 15.2%. The correction baseline is also fairly high – 12.4%. The problem is more difficult since typically the system performs more than one unplanned implicit confirmation in a single turn. As a result, even if disconfirmation markers are available in the user response, we are facing the problem of detecting which of the implicitly confirmed concepts the user is trying to correct. Still, the proposed data-driven models significant outperform both the heuristic and the correction baseline.

We tested the statistical significance of the improvements produced by the proposed models. The results indicate that the improvements over the heuristic baselines are significant (both in terms of classification error and Brier score) at levels below $p<0.003$ (in most cases below 10^{-5}). The belief updating models for implicit and unplanned implicit confirmations also significantly outperform the correction baseline ($p<0.001$). For these actions, the full model significantly outperformed the basic model, indicating the usefulness of prior and confusability features. Finally, no statistically significant differences could be observed at a p-level of 0.05 between the full and runtime models.

6.4.4.4 Model analysis

We inspected the resulting logistic regression models (see Table 28), in an effort to better understand which features were most informative for the belief updating tasks. The coefficients for each feature describe the effect of the feature on the log-odds of the "other" hypothesis being correct, versus the initial hypothesis h1. A negative coefficient indicates that the feature that increases the likelihood of the initial hypothesis being correct; a positive coefficient indicates a feature that decreases that likelihood.

For explicit confirmations, the presence of positive or negative confirmation markers (`answer_type=YES`, `answer_type=NO`, `answer_type=other`) is informative, as expected. The confusability score of the initial hypothesis (`i_h1_confusability`) is also informative, and the larger the score, the more likely that the initial hypothesis is correct (recall that higher values of the confusability score mean the value is less confusable). On the other hand, the presence of a new hypothesis for that concept in the recognition result (`srh_r1_avail`) indicates that the initial hypothesis is less likely to be correct. The model also learned that, when the recognized user response consists of the single word "small" (`lex_w1:SMALL`), the initial hypothesis is most likely incorrect. This happens because *"no"* is often misrecognized as *"small"* (the two words are acoustically similar). Finally, the last feature in this model indicates that, when the concept confirmed is equip (`concept_id:equip`), the initial hypothesis is most likely incorrect.

Similarly, for implicit confirmations, the presence of lexical confirmation markers (`answer_type`), the initial confusability (`i_h1_confusability`), as well as the presence of a new concept value in the recognition result (`srh_r1_avail`) plays an important role. Another feature selected by this model is the initial confidence score of the confirmed hypothesis (`i_h1_confidence`):

EC (explicit confirmation): full model		
Feature	Coef.	Effect
k	4.33	+
answer_type=YES	-5.04	-
answer_type=NO	2.87	+
answer_type=other	-1.00	-
i_h1_confusability	-4.83	-
srh_r1_avail	2.87	+
perc_unvoiced>m	-1.79	-
lexw1:SMALL	34.43	+
concept_id:equip	4.51	+

IC (implicit confirmation): full model		
Feature	Coef.	Effect
k	3.64	+
i_h1_confusability	-4.49	-
answer_type=YES	-1.71	-
answer_type=NO	1.59	+
srh_r1_avail	3.30	+
lex:THREE	-2.69	-
i_h1_confidence	-3.49	-
turn	0.03	+

Table 28. $BU_{n=0}^{m=1}$ belief updating models for explicit and implicit confirmations

the higher the initial confidence score, the more likely that the initial hypothesis is correct. Finally, the model also includes information about the turn number and the presence of the word "*three*" in the recognized user response.

The features selected and their weights correspond largely to our intuitions about the belief updating process. In fact, a number of these features are already used by our heuristic belief updating rule (the lexical confirmation markers, the initial confidence score, and the presence of a new value in the follow-up user response). However, the data-driven models discovered a number of additional informative features (e.g. contextual, prosodic, lexical, confusability), and tuned their weights to optimize the accuracy of the resulting beliefs.

6.4.5 The $BU_{n=1}^{m\geq 1}$ model: updating beliefs after all system actions

6.4.5.1 Model

The $BU_{n=0}^{m=1}$ model described in the previous section did not incorporate any new hypotheses from the recognition result into the belief space (n=0). In this section, we remove this restriction and report results on the more general model family: $BU_{n=1}^{m\geq 1}$. These models keep track of m initial hypotheses, and add one new hypothesis from the recognition result in each belief updating step. Since these models can incorporate new values from the user response, they can be used in conjunction with any system action, including **request**, and **no-action.** (In contrast to the $BU_{n=0}^{m=1}$ model could only be used following confirmation actions). Note that we are still using a single recognition hypothesis at any given time (n=1).

We report results on 2 different models: $BU_{n=1}^{m=1}$ and $BU_{n=1}^{m=2}$. In the first case, the multinomial output variable e_{t+1} has degree 3: $e_{t+1}=<p_{h1}, p_{r1}, p_{other}>$. In the second case, it has degree 4: $e_{t+1}=<p_{h1}, p_{h2}, p_{r1}, p_{other}>$. For instance, the equations governing the first model are:

$$P(C = S_t\{h1\} \mid \bar{F}) = p_{h1} = \frac{e^{\overline{\alpha_{h1}} \cdot \bar{F}}}{1 + e^{\overline{\alpha_{h1}} \cdot \bar{F}} + e^{\overline{\alpha_{r1}} \cdot \bar{F}}}$$

$$P(C = S_t\{r1\} \mid \bar{F}) = p_{r1} = \frac{e^{\overline{\alpha_{r1}} \cdot \bar{F}}}{1 + e^{\overline{\alpha_{h1}} \cdot \bar{F}} + e^{\overline{\alpha_{r1}} \cdot \bar{F}}}$$

$$P(C = \text{other} \mid \bar{F}) = p_{other} = \frac{1}{1 + e^{\overline{\alpha_{h1}} \cdot \bar{F}} + e^{\overline{\alpha_{r1}} \cdot \bar{F}}}$$

6.4.5.2 Features

The same set of features $\bar{F}$ described in the previous section (see Table 26) was used. Again, one separate belief updating model was constructed for each of the five possible system actions.

6.4.5.3 Empirical results

§ Results for $BU_{n=1}^{m=1}$ model

We begin by discussing the results for the $BU_{n=1}^{m=1}$ model. The training and evaluation procedure was the same as the one described in the previous section. The results are presented in Table 29 and illustrated in Figure 77.

For the request action, the initial baseline is uninformative. Prior to engaging in a request, the concept is empty, and therefore the initial hypothesis (empty) will most often be incorrect (98.2%). In a few remaining cases (1.8%), the initial empty value is indeed correct since the user does not actually specify a value for the requested concept in the follow-up response. The heuristic belief updating rule constructs an updated belief by taking into account the value heard from the recognizer, with the corresponding confidence score. This leads to a classification error of 9.5%, which

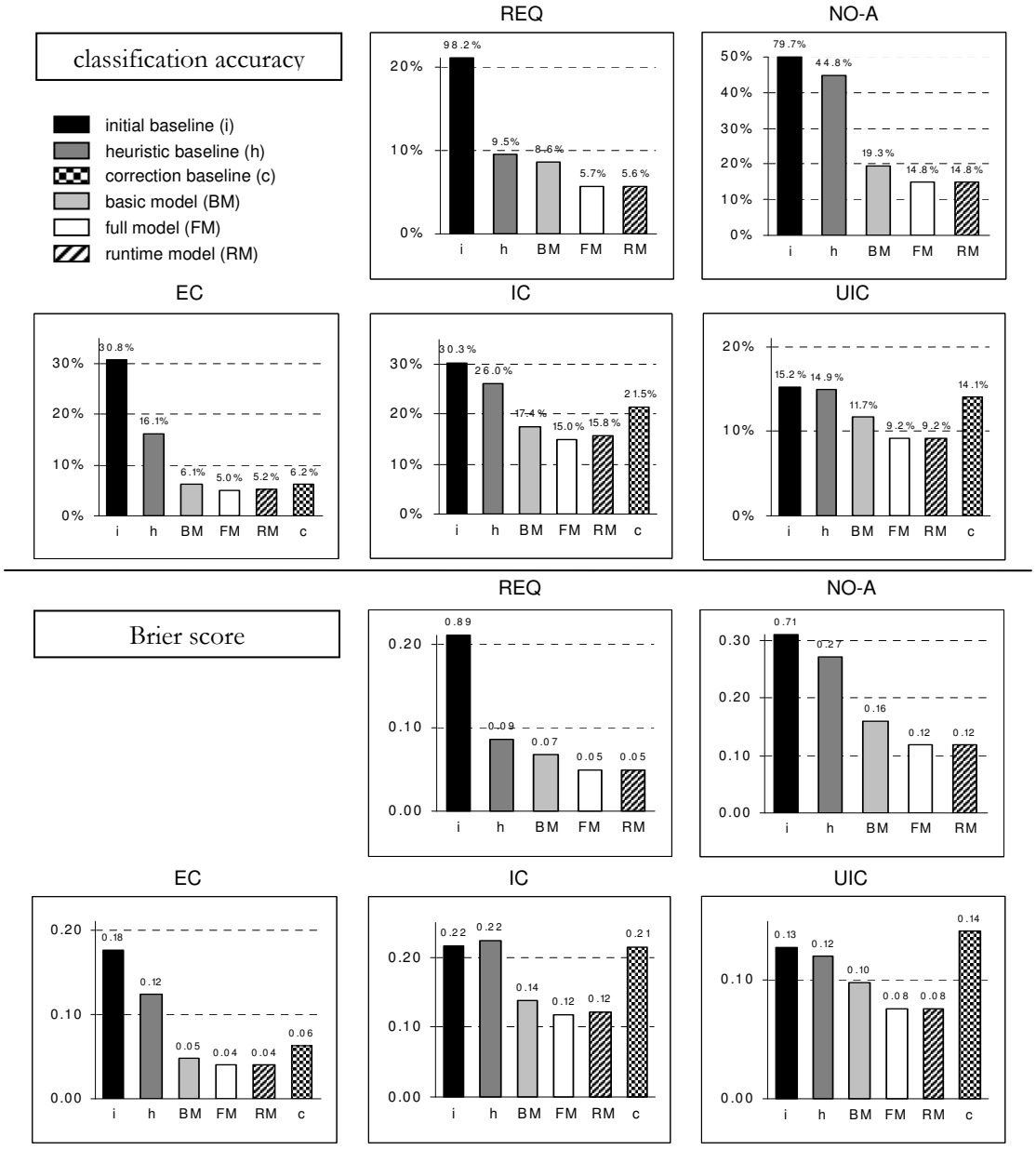


Figure 77. Performance of $BU_{n=1}^{m=1}$ belief updating models

Action	Baselines			Models		
	initial	heuristic	correction	basic	full	runtime
REQ (classif. error)	98.2%	9.5%	-	8.6%$^{\varnothing}$	5.7%	5.6%
(Brier score)	0.8860	0.0862	-	0.0672	0.0489	0.0491
NO-A (classif. error)	79.7%	44.8%	-	19.3%	14.8%	14.8%
(Brier score)	0.7094	0.2704	-	0.1603	0.1189	0.1189
EC (classif. error)	30.8%	16.1%	6.2%	6.1%	5.0%	5.2%
(Brier score)	0.1755	0.1240	0.0619	0.0478	0.0409	0.0401
IC (classif. error)	30.3%	26.0%	21.5%	17.4%	15.0%	15.8%
(Brier score)	0.2168	0.2238	0.2145	0.1373	0.1168	0.1209
UIC (classif. error)	15.2%	14.9%	14.1%	11.7%	9.2%	9.2%
(Brier score)	0.1273	0.1205	0.1412	0.0987	0.0759	0.0759

Table 29. Performance of $BU_{n=1}^{m=1}$ belief updating models (all results except for the one marked $^{\varnothing}$ represent statistically significant improvements over the heuristic baseline; results in bold face represent statistically significant improvements over the correction baseline)

constitutes our heuristic baseline. The correction baseline is not defined because in this case the system did not engage in a confirmation action. The basic belief updating model leads to a small improvement over the heuristic (8.6% versus 9.5%), which is not statistically significant. Our confidence annotator is well-tuned to this domain, and the proposed belief updating model cannot significantly improve performance above that (when it uses a very similar feature set.) However, once prior and confusability features are added, a significant performance gain can be observed: 5.7% for the full model. The runtime model performs similarly at 5.6%. These results highlight again the usefulness of value-specific features such as priors and confusability. The Brier score evaluation (see Table 29) reveals a similar result.

For the **no-action** case, the initial baseline (79.7%) is again uninformative because in a large number of cases the initial value for the concept is empty. Like in the request case, the correction baseline is not defined. The heuristic belief updating rule leads to an error rate of 44.8%. All models show improvements over the heuristic rule: 19.3% for the basic model and 14.8% for the full and runtime models. The full models (which include priors and confusability) significantly outperform the basic models.

For the remaining three actions – explicit confirmation, implicit confirmation, and unplanned implicit confirmation – the results are very similar to previous results obtained with the $BU_{n=0}^{m=1}$ model. The $BU_{n=1}^{m=1}$ model creates beliefs that are significantly more accurate than the ones constructed by the heuristic rule, and even than the correction baseline. The improvements in Brier score are statistically significant across the board.

The small differences in the initial, heuristic and correction baselines between Table 29 and Table 27 are explained by the fact that the models operate over different belief spaces. For the $BU_{n=0}^{m=1}$ model, the target variable is binomial, so the classification error and Brier scores correspond to a 2-class problem. For the $BU_{n=1}^{m=1}$ model, the target variable is a multinomial of degree 3, and the classification error and Brier score correspond to a multiple (3) class classification problem.

§ Results for $BU_{n=1}^{m=2}$ model

In general, the $BU_{n=1}^{m=2}$ belief updating model performed similarly to the $BU_{n=1}^{m=1}$ model. The results are illustrated in Table 30. A comparison of training and test set performance revealed that for some actions, shown in boldface in Table 30, the $BU_{n=1}^{m=2}$ models did over-fit to the training data. A closer investigation of these models showed that the cause is data sparsity. The multinomial target variable for the $BU_{n=1}^{m=2}$ model has degree 4: the underlying belief space is {h1, h2, r1, other}. The h2 class is very poorly represented in the training set. In only 151 out of the 11332 training instances the h2 hypothesis is non-empty, i.e. the initial belief contains a second concept hypothesis. Furthermore, in only 40 of these cases h2 is the correct class, i.e. the second initial hypothesis is the correct hypothesis. As a consequence, very few or zero observations exist for certain combinations of feature values

Action		Baselines initial	Baselines heuristic	Baselines correction	Models basic	Models full	Models runtime
REQ (c.e.)	BIC	98.2%	9.5%	-	8.6%	5.7%	5.6%
(Brier)		0.8860	0.0862	-	0.0672	0.0489	0.0491
	CV				8.6%	5.9%	5.9%
					0.0667	0.0499	0.0499
NOA (c.e.)	BIC	79.7%	44.8%	-	20.1%	21.1%	21.1%
(Brier)		0.7094	0.2704	-	0.1669	0.1887	0.1887
	CV				22.2%	19.8%	19.8%
					0.1851	0.1499	0.1499
EC (c.e.)	BIC	30.9%	16.2%	6.2%	6.1%	5.0%	5.5%
(Brier)		0.1757	0.1242	0.0619	0.0491	0.0439	0.0445
	CV				8.1%	5.6%	6.1%
					0.0537	0.0461	0.0479
IC (c.e.)	BIC	30.3%	26.0%	21.5%	29.0%	16.0%	16.2%
(Brier)		0.2168	0.2238	0.2145	0.2548	0.1308	0.1300
	CV				19.8%	16.3%	16.4%
					0.1568	0.1317	0.1334
UIC (c.e.)	BIC	15.2%	14.9%	14.1%	11.6%	9.2%	9.2%
(Brier)		0.1273	0.1205	0.1412	0.0980	0.0771	0.0771
	CV				11.8%	9.8%	9.8%
					0.1013	0.0810	0.0810

Table 30. Performance for the $BU_{n=1}^{m=1}$ belief updating models (using BIC and Cross-Validation for regularization)

and the h2 output class. In some cases those features appear as very informative, but this conclusion is based on a very small number of samples, and hence often invalid. We suspect these problems might be alleviated if more training data were available.

Given the limited amount of available training data, a more conservative approach was used to prevent over-fitting: instead of using the Bayesian Information Criterion, we stopped adding features to the model as soon as the average data log-likelihood on the validation set decreased. The results using this new cross-validation regularization method are also shown side by side with the initial results in Table 30. The new models do not over-fit the training data.

The classification error rates and Brier scores for the models discussed so far ($BU_{n=1}^{m=2}$ in Table 30, $BU_{n=1}^{m=1}$ in Table 29, and $BU_{n=0}^{m=1}$ in Table 27) are not directly comparable since the underlying belief spaces for these models are different. A comparative performance analysis of these models, using a common evaluation metric, is discussed later, in section 6.4.7.

6.4.5.4 Model analysis

Next, we inspected the learned models in order to identify the most informative features. The full set of $BU_{n=1}^{m=1}$ models is presented in Appendix B. For brevity purposes we only discuss here the model for the no-action system action – see Table 31. Recall that this is the case in which the system hears a value for the concept, without specifically asking for that concept or engaging in any confirmation action with respect to it.

The model coefficients are somewhat more difficult to interpret in this multinomial model. The r1/h1 coefficient reflects the log-odds for the new hypothesis (r1) being correct versus the initial top hypothesis (h1); in other words, how many times more likely is the new hypothesis versus the old hypothesis (on a log scale). A positive coefficient indicates that a higher feature value will increase the likelihood of the new hypothesis versus the old hypothesis; a negative coefficient indicates the opposite. The other/r1 coefficient indicates the log-odds for the other hypothesis being correct, i.e. neither h1 or r1 being correct, versus the initial hypothesis being correct.

As Table 31 shows, the model selects a fairly large number of features (22 in this case) from different knowledge sources in the system. The model uses prior and confusability information about both the initial and the new hypothesis, as well as other information such as the number of words in the recognized results (`word_num`), the identity of the concept confirmed (`concept_id`), whether the

NOA (no action): full model		
	Coefficients	
Feature	r1/h1	other/h1
k	1.79	6.90
srh_r1_confusability	5.17	-0.26
ivs=value	-2.96	-1.79
ivs=empty_no_hist	-2.33	-3.43
word_num=1	-2.37	-0.04
word_num=2	-0.22	0.16
word_num=3	0.12	0.31
i_h1_prior	-0.58	-0.99
concept_id:date	0.77	6.43
i_h1_prior_gt_1	0.89	-3.45
srh_r1_explicitly_disconfirmed_already	-5.89	1.33
concept_id:_ChooseAnyRoom_trigger	16.31	3.37
i_h1_confusability	-4.52	-3.77
ih_diff_lexical	-1.15	-0.51
srh_r1_prior	0.28	-0.02
srh_h_h1_avail	-1.68	-4.05
lex:THAT'S	1.21	2.52
concept_id:size	0.80	8.15
dialog_state_id:HowMayIHelpYou	-0.25	-1.59
h_avg_confidence	3.25	1.29
i_h1_explicitly_confirmed_already	-13.68	-14.84
word_num	0.40	0.03
lm_score	0.00	0.00

Table 31. $\mathbf{BU}_{n=1}^{m=1}$ belief updating model for no-action.

initial hypothesis or the new hypothesis had already been confirmed or disconfirmed, lexical information (`lex`), language model information (`lm_score`), etc.

The precise contribution of each of these features is determined based on the training data. In most cases, the coefficients confirm the intuition. For instance if the new hypothesis heard had already been disconfirmed previously in the dialog (`srh_r1_explicitly_disconfirmed_already`), then its likelihood drops with respect to the initial hypothesis (-5.89). At the same time, the likelihood of the "other" hypothesis is slightly increased. On the other hand, if the initial hypothesis had already been confirmed (`srh_h1_explicitly_confirmed_already`), then the likelihood of the new hypothesis drops dramatically (-13.68), but so does the likelihood of the "other" hypothesis (-14.84).

In summary, the belief updating models take into account many other knowledge sources in comparison with current heuristics. The informative features are automatically selected from a large pool of features, and the model parameters are optimized accordingly.

6.4.5.5 Scalability: an analysis of belief updating performance vs. concept cardinality

Next, we investigated the scalability of the proposed belief updating models. Clearly, the proposed approach scales well with the number of concepts in the system. In fact, since the models update each concept independently, the proposed approach can be used in spoken dialog systems operating with an unlimited number of concepts. Information about concept identity can be used as a feature in the models, but a separate model does not need to be trained for every concept the system operates with. Instead, the compressed, abstracted belief representation allows us to use the entire corpus of concept updates in order to train belief updating models that generalize across concepts.

Nevertheless, a second scalability issue remains: how well do the proposed models scale with the cardinality of the concept? What is the model performance for concepts with a small versus large numbers of possible values? In Figure 78 we show the $\mathbf{BU}_{n=1}^{m=1}$ model performance (obtained in cross-validation, and averaged across different system actions) as a function of concept cardinality. With the notable exception of the Boolean concepts, the models perform similarly, regardless of the cardinality of the concept, both in terms of classification error and Brier score. For instance the model

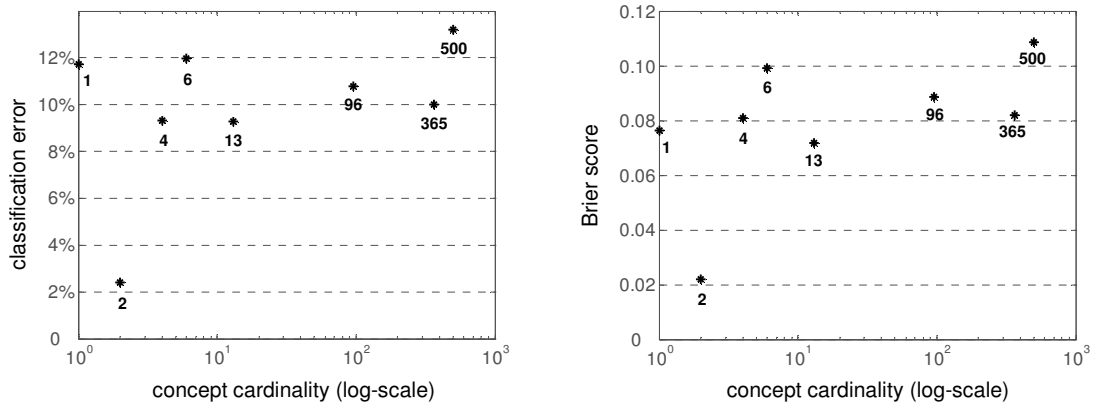


Figure 78. Average performance for the $BU_{n=1}^{m=1}$ model, as a function of concept cardinality

performance for the `start_time` concept (cardinality 96), `date` concept (cardinality 365) and `room_size` concept (cardinality 500) is very similar to the model performance for the `room_equip` concept (cardinality 4), `room_size_spec` concept (cardinality 6), or `room_location` concept (cardinality 13). We believe the better performance for the Boolean concepts is in part explained by better speech recognition performance for the possible values for this concept, i.e. yes and no.

6.4.5.6 Sample efficiency: an analysis of training set size requirements

Next, we investigated the relationship between model performance and training set size. Like for confidence annotation, we are interested in finding out how sample efficient the proposed models are, i.e. what are the training set size requirements?

Successive models were trained using increasingly larger amounts of data: 20 random permutation of each dataset were generated, and 200 data-points were held out for testing. The $BU_{n=1}^{m=1}$

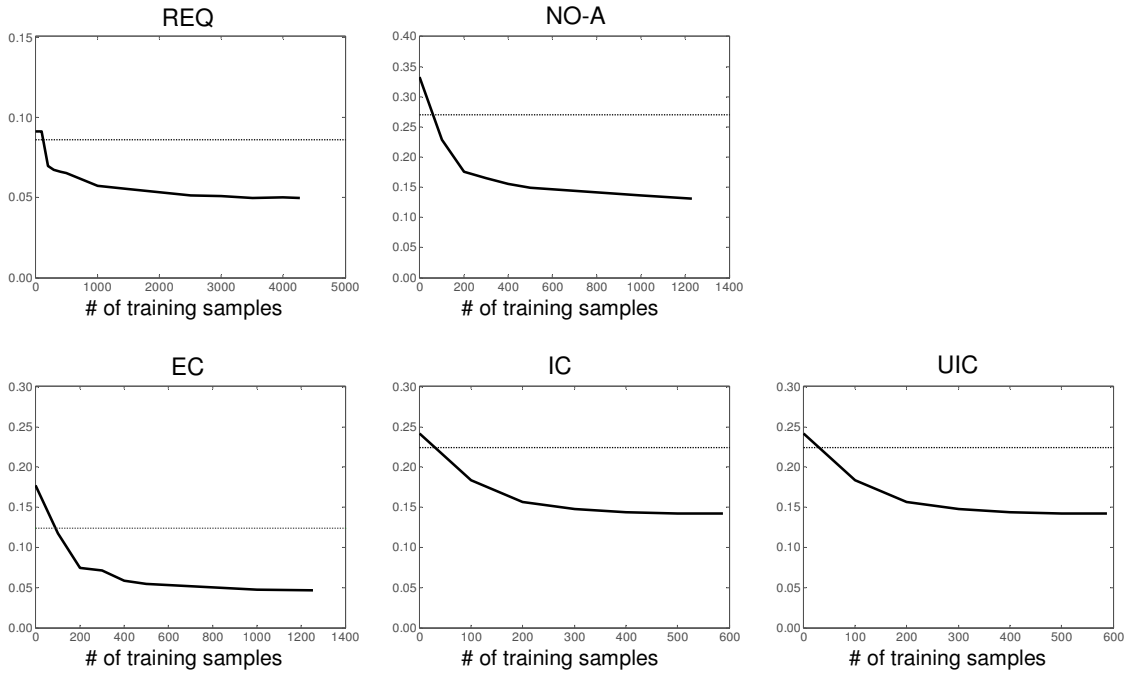


Figure 79. Brier score for $BU_{n=1}^{m=1}$ model as a function of increased training set size (dotted line shows Brier score for heuristic belief updating rule)

models were trained using 100, 200, 300, 400, 500, 1000 samples, and so forth, continuing in increments of 500. The models were evaluated on the 200 held out data-points.

The Brier score results for the 5 models corresponding to each system action are illustrated in Figure 79 (the classification error results reveal a very similar pattern). The performance of the models seems to have reached an asymptote for all system actions. The results from Figure 79 indicate in fact that the models could be trained to attain very similar performance with only a third of the available training data, i.e. with only 150 dialogs.

6.4.6 The $BU_{n\geq 1}^{m\geq 1}$ model: considering multiple recognition hypotheses

6.4.6.1 Model

The $BU_{n=1}^{m\geq 1}$ belief updating models described in the previous section added at most one additional hypothesis from the recognition result in each belief updating step (n=1). In this section we remove this restriction and consider the more general class of models $BU_{n\geq 1}^{m\geq 1}$. These models allow the system to integrate multiple (n>1) concept-level hypotheses from each input, for instance by using a recognition n-best list.

The Sphinx-II speech recognition engine used in these experiments is not able to perform the n-best list computation fast enough to sustain a real-time end-to-end interaction. As a consequence, we could not obtain and/or use this type of information at runtime. Recall however that the RoomLine system is equipped with two parallel Sphinx-II recognition engines, using gender-specific acoustic models. Each of these engines provides a separate recognition hypothesis; selection is performed later, based on the utterance-level confidence score assigned by Helios. The availability of these two hypotheses allowed us to develop and evaluate $BU_{n=2}^{m\geq 1}$ models.

In particular, in this section, we report on a $BU_{n=2}^{m=1}$ model. The model updates the belief space by retaining the previous topmost hypothesis, and adding at most two new hypotheses (r1, r2) from the user response; the user response consists in this case of the two recognition hypotheses generated by the parallel Sphinx-II engines. The multinomial output variable in this model, e_{t+1}, has degree 4: $e_{t+1}=<p_{h1}, p_{r1}, p_{r2}, p_{other}>$. The equations that govern the model are:

$$P(C = S_t\{h1\} \mid \bar{F}) = p_{h1} = \frac{e^{\overline{\alpha_{h1}}\cdot\bar{F}}}{1 + e^{\overline{\alpha_{h1}}\cdot\bar{F}} + e^{\overline{\alpha_{r1}}\cdot\bar{F}} + e^{\overline{\alpha_{r2}}\cdot\bar{F}}}$$

$$P(C = S_t\{r1\} \mid \bar{F}) = p_{r1} = \frac{e^{\overline{\alpha_{r1}}\cdot\bar{F}}}{1 + e^{\overline{\alpha_{h1}}\cdot\bar{F}} + e^{\overline{\alpha_{r1}}\cdot\bar{F}} + e^{\overline{\alpha_{r2}}\cdot\bar{F}}}$$

$$P(C = S_t\{r2\} \mid \bar{F}) = p_{r2} = \frac{e^{\overline{\alpha_{r2}}\cdot\bar{F}}}{1 + e^{\overline{\alpha_{h1}}\cdot\bar{F}} + e^{\overline{\alpha_{r1}}\cdot\bar{F}} + e^{\overline{\alpha_{r2}}\cdot\bar{F}}}$$

$$P(C = \text{other} \mid \bar{F}) = p_{other} = \frac{1}{1 + e^{\overline{\alpha_{h1}}\cdot\bar{F}} + e^{\overline{\alpha_{r1}}\cdot\bar{F}} + e^{\overline{\alpha_{r2}}\cdot\bar{F}}}$$

6.4.6.2 Features

To construct this model, we started with the same set of features used in the previous models (described in Table 26), and added a number of additional features that capture aspects of he extended user response. The additional features are described in more detail in Table 32.

6.4.6.3 Empirical results

The results for the $BU_{n=2}^{m=1}$ model are shown in Table 33. As with the $BU_{n=1}^{m=2}$ models, simply using the Bayesian Information Criterion is not sufficient to always prevent over-fitting in these models. The belief space for the $BU_{n=2}^{m=1}$ model has degree 4: the underlying belief space is {h1, r1, r2, other}. In

Feature name	Type	Derived features	Feature description
Features characterizing new values in the user response (the two parallel recognition hypothesis)			
ur2_i_h1_avail	B		the initial top concept hypothesis is present in the user response (i.e. in at least one of the 2 parallel hypotheses)
ur2_i_h1_confidence	C		the confidence score for the top concept hypothesis in the user response
ur2_r1_avail	B		the user response contains a new concept hypothesis
ur2_r1_confidence	C	>.25, >.50, >.75	the confidence score for the first new concept hypothesis contained in the user response
ur2_r2_avail	B		the user response contains a second new concept hypothesis
ur2_r2_confidence	C	>.25, >.50, >.75	the confidence score for the second new concept hypothesis contained in the user response
ur2_r1_diff_r2	B		the two parallel hypotheses contain two different new concept values
Priors			
ur2_r1_prior	C	>1	prior score for the first new concept hypothesis in the user response
ur2_r2_prior	C	>1	prior score for the second new concept hypothesis in the user response
Confusability			
ur2_r1_confusability	C	>m	confusability score for the first new concept hypothesis in the user response
ur2_r2_confusability	C	>m	confusability score for the second new concept hypothesis in the user response

Table 32. Additional belief updating features for the $\mathbf{BU}^{m=1}_{n=2}$ model

Action		Baselines initial	Baselines heuristic	Baselines correction	Models basic	Models full	Models runtime
REQ (c.e.) (Brier)	BIC	98.2% 0.8860	9.5% 0.0862	- -	8.8% 0.0741	6.2% 0.0531	6.2% 0.0535
	CV				8.7% 0.0718	5.8% 0.0505	5.8% 0.0505
NOA (c.e.) (Brier)	BIC	79.7% 0.7094	44.8% 0.2704	- -	22.1% 0.1799	24.8% 0.2111	17.0% 0.1356
	CV				23.5% 0.1815	18.2% 0.1439	18.2% 0.1439
EC (c.e.) (Brier)	BIC	30.9% 0.1757	16.2% 0.1242	6.2% 0.0619	11.8% 0.0894	10.5% 0.0790	15.1% 0.1077
	CV				5.7% 0.0441	5.1% 0.0415	5.8% 0.0437
IC (c.e.) (Brier)	BIC	30.3% 0.2168	26.0% 0.2238	21.5% 0.2145	26.5% 0.1793	18.8% 0.1424	18.8% 0.1449
	CV				21.7% 0.1610	17.1% 0.1287	17.1% 0.1287
UIC (c.e.) (Brier)	BIC	15.2% 0.1273	14.9% 0.1205	14.1% 0.1412	13.2% 0.1082	10.4% 0.0811	10.4% 0.0811
	CV				12.0% 0.1005	9.9% 0.0790	9.9% 0.0790

Table 33. Performance for the $\mathbf{BU}^{m=1}_{n=2}$ belief updating models (using BIC and Cross-Validation for regularization)

this case, the r2 class is under-represented in the training data. The r2 hypothesis is non-empty in only 327 of the total 11332 number of training points; in other words, in only 327 instances the system can extract two new different concept hypotheses from the two parallel recognition hypotheses. And only in 83 of these instances the second new hypothesis (r2) is the correct one. As Table 33 shows, using a cross-validation regularization method (as opposed to the Bayesian Information Criterion) also helps in these cases.

6.4.7 A comparative evaluation of belief updating models

In the previous sections we have described and evaluated four incrementally more complex belief updating models. The first model, $BU_{n=0}^{m=1}$, reasons only about the top initial hypothesis of a concept and updates the system's belief in that value in light of user responses to system confirmation actions. The second model, $BU_{n=1}^{m=1}$, keeps track of at most two concept hypotheses: in each belief updating step, it keeps the current top concept hypothesis, adds a new hypothesis from the follow-up user response, and constructs an updated belief over this space. The third and fourth models keep track of at most three concept hypotheses. The third model, $BU_{n=1}^{m=2}$, operates with a belief space formed top two initial concept hypotheses and a new hypothesis from the recognizer. The fourth model, $BU_{n=2}^{m=1}$, operates with a belief space formed of the top initial concept hypothesis and two new concept hypotheses extracted from two parallel recognition results; in the more general case, these hypotheses could be exatracted from an n-best list.

It is interesting to understand how these models compare to each other. The classification-error-rates and Brier scores presented so far for these models in Table 27, Table 29, Table 30 and Table 33 are not directly comparable to each other because each model operates and is evaluated in a different underlying belief space. To perform a valid comparison, we expanded the compressed beliefs constructed by each model into their corresponding full beliefs. This was accomplished by uniformly dividing the probability mass associated with the "other" hypothesis among the remaining possible concept values. The Brier scores were then re-computed in this full belief space. The results are presented in Table 34 and illustrated in Figure 80.

The results indicate that the three models that take into account at least one new hypothesis from the follow-up user turn, $BU_{n=1}^{m=1}$, $BU_{n=1}^{m=2}$, $BU_{n=2}^{m=1}$, perform better than the simplest model $BU_{n=0}^{m=1}$. This result is expected: ignoring new concept hypotheses in the user response is clearly detrimental. In addition, there are no significant differences between the performance of the last three models: increasing the value of k beyond 2 does not lead to additional performance gains. This result is also not surprising because there are only few cases in the training data in which there was a valuable second hypothesis in the initial belief or in the user response. We believe that this result is to a large extent an artifact of the particular recognition system and experimental setup we have used. In

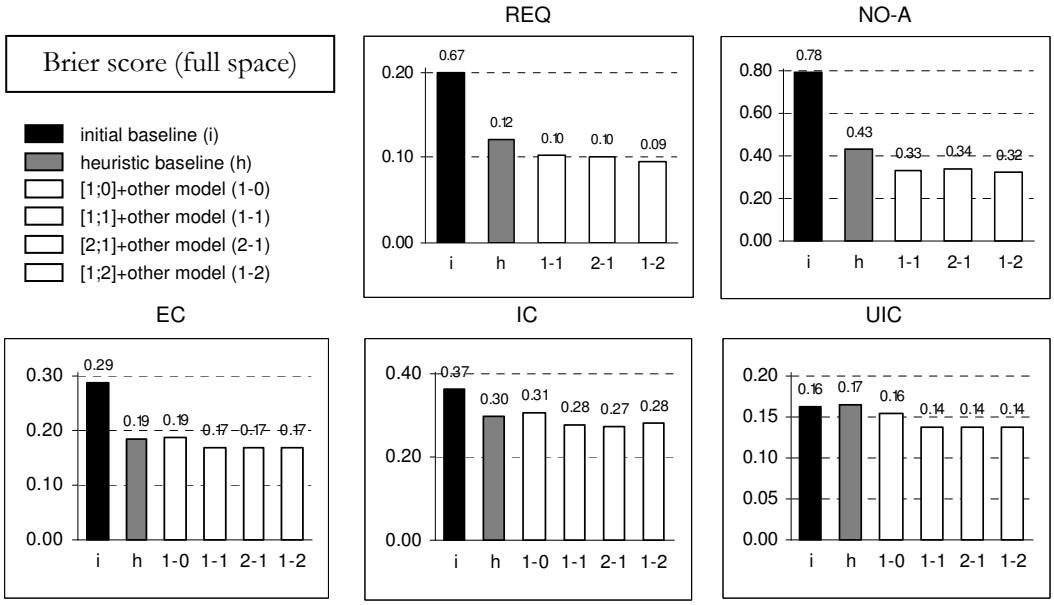


Figure 80. Performance comparison between $BU_{n=0}^{m=1}, BU_{n=1}^{m=1}, BU_{n=1}^{m=2}, BU_{n=2}^{m=1}$ belief updating models

Action	Baselines		Models			
	initial	heuristic	$BU^{m=1}_{n=0}$	$BU^{m=1}_{n=1}$	$BU^{m=2}_{n=1}$	$BU^{m=1}_{n=2}$
REQ	0.6711	0.1212	-	0.1019	0.1014	0.0949
NO-A	0.7872	0.4259	-	0.3264	0.3389	0.3249
EC	0.2862	0.1851	0.1870	0.1693	0.1675	0.1696
IC	0.3652	0.2991	0.3076	0.2766	0.2731	0.2820
UIC	0.1637	0.1650	0.1557	0.1373	0.1380	0.1377

Table 34. Performance comparison between $BU^{m=1}_{n=0}$, $BU^{m=1}_{n=1}$, $BU^{m=2}_{n=1}$, $BU^{m=1}_{n=2}$ belief updating models

general, we conjecture that the more complex models could leverage an n-best list that contained multiple alternate concept-level hypotheses for further performance gains. In the future, it would be interesting to understand how the proposed belief updating models perform under those conditions.

6.4.8 Concluding remarks

Spoken dialog systems typically use confidence scores to detect potential misunderstandings. While confidence scores can provide an initial, turn- or concept-level assessment of the reliability of the information obtained from the recognizer, ideally systems should combine evidence from subsequent user turns to update their beliefs throughout the conversation.

In this section, we have discussed a data-driven approach for this belief updating problem. The proposed approach builds upon previous work on confidence annotation and correction detection and provides a unified framework that allows a spoken dialog system to integrate evidence from multiple turns in the conversation and continuously update and improve the accuracy of its beliefs. The approach uses a compressed representation of beliefs: instead of storing and updating a probability distribution over the full set of possible values for a concept, the proposed belief updating models keep track only of the top-k most likely hypotheses (where k is a small integer). In each belief updating step, the system remembers the top-m of the initial top-k hypotheses for a concept, and adds n new hypotheses from the recognition results. The underlying compressed belief space for each concept changes therefore throughout the conversation, as new concept hypotheses are added and old ones are dropped. This compressed belief representation allows us to cast the 1-step belief updating problem as a multinomial regression task, and makes a learning-based approach tractable.

Using the proposed methodology, we have constructed and evaluated four different belief updating models operating with increasingly larger belief spaces (from k=1 to k=3). Empirical results show that the proposed models significantly outperform typical heuristic rules used for this task. In some cases, the models perform even better than a heuristic that has access to a perfect correction detector. Additionally, we have analyzed the sensitivity of these models to the amounts of training data and to the cardinality of the concepts they operate over. The results indicate that the models are both sample efficient and scalable.

The proposed belief updating models select and combine a large number of features extracted from different knowledge sources in the system. The relative weights of these features are learned automatically from data; often, the selected features and their weights are in line with our intuitions. Experiments with different feature subsets have highlighted the importance of high-level and domain-specific information such as the identity of the concept to be updated, the prior likelihoods of various concept values, and confusability scores. We believe that further improvements in performance can be attained by leveraging additional high-level features. For instance, in these experiments the prior likelihoods were constructed manually by a domain expert, for only 3 of the 28 concepts that the system operates with. Data-driven priors might provide an even more accurate basis for belief updating. Other high-level knowledge such as domain-specific constraints or inter-concept dependencies may also lead to further performance improvements. For instance, no conference room reservations are made from 4 p.m. to 2 p.m.; rather, reservations are made from 2 p.m. to 4 p.m. This type of domain-specific constraint could also be captured as a feature and used in the belief updating process.

In these experiments, the models operating over larger belief spaces, e.g. using k=3, did not produce significant improvements over the simpler k=2 models. We believe this is an artifact of the particular setup in which the training data was collected: no n-best list information was available. If the speech recognizer can generate a dense concept-level n-best list, further performance improvements might be possible by using models with larger values of n, i.e. models that incorporate in the belief space multiple hypotheses from the n-best list in each turn.

The model evaluations discussed so far were local in character. While the results are encouraging, our ultimate goal is not just to construct accurate beliefs, but rather to improve the efficiency and effectiveness of the interaction by doing so. One important question therefore remains: will the observed improvements on the one-step belief updating task lead to significant improvements in global dialog performance? We address this question in the next section.

6.5 Impact on global dialog performance

To assess the impact of the proposed belief updating models, we conducted an additional user study in which we compared a system that used the heuristic belief updating rules against a system that used the proposed data-driven belief updating models. We begin by describing the experimental design in the next subsection. Then, in subsection 6.5.2.1, we discuss in detail the results of this experiment.

6.5.1 Experimental design

The user study was designed as a between-groups experiment, during which 40 participants interacted with the RoomLine system. The participants were randomly assigned into 2 gender-balanced groups: **control** and **treatment**. Participants in the **control** group interacted with a version of the system that used simple heuristic update rules (described more extensively in section 6.4.2.4) to perform belief updates. Participants in the **treatment** group interacted with a version of the system that used the runtime belief updating models. In all other respects, the two systems were identical.

During the experiment, each participant attempted a maximum of 10 scenario-based interactions with the system, within a set time of 40 minutes. The same 10 scenarios were presented in the same order to all participants. The scenarios were designed to cover all the important aspects of the system's functionality and had different degrees of difficulty. To avoid language entrainment, the

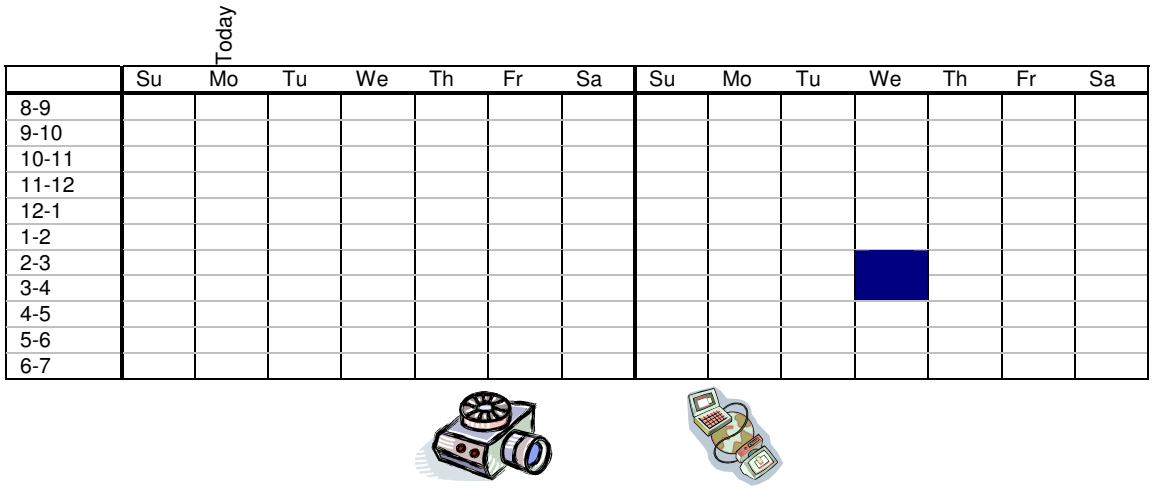


Figure 81. Sample room reservation scenario

scenarios were presented to the participants in a graphical fashion. An example scenario is shown in Figure 81. In this example, the user had to reserve a conference room with a projector and a network connection on the following Wednesday, from 2 p.m. to 4 p.m. The graphical encoding was explained to the participants prior to the experiment.

All the participants in this experiment were non-native speakers of north-American English; none of them had any previous experience with the RoomLine system. This choice was motivated by the intuition that improvements in belief updating performance are likely to translate into overall performance improvements especially when if the word-error-rate is high. At low word-error-rates, simply trusting the inputs leads to generally accurate beliefs. Not many opportunities for improvement exist in this case. However, as recognition performance degrades, it becomes more and more important that the system is able to accurately assess its beliefs. In this case, an accurate belief updating mechanism has the potential to significantly improve dialog performance. This hypothesis was indeed confirmed by empirical evidence; we discuss it in more detail in section 6.5.2.

6.5.2 Empirical results

6.5.2.1 Data

The corpus collected in this experiment contains a total of 384 dialog sessions and 6389 user turns. Each user turn was orthographically transcribed by a human annotator and the turn-, session- and user-level word-error-rates were computed. A binary measure of task success was also computed.

One of the 40 participants systematically misunderstood 7 out of the 10 scenarios. We excluded all the data collected from this participant from the analysis presented below. Keeping the corresponding data in the corpus leads to a stronger, but we believe less accurate result.

6.5.2.2 Impact on effectiveness

We first investigated the impact of the experimental condition on the effectiveness of the interaction, i.e. on the task success rate. The overall task success rate in the treatment condition was larger than in the control condition: 81.3% versus 73.7% ($p=0.0724$).

Apart from the experimental condition, another factor that exerts a very significant effect on dialog performance is the recognition accuracy. As discussed earlier we expected improvements especially at higher word-error-rates (WER). As a first step towards better understanding the effect of the belief updating models at different word-error-rates we binned the sessions in 6 classes according to their average word-error-rate and then computed the average task success rates in the treatment and control conditions for each class. The results are shown in Figure 82. The plot seems to confirm that indeed larger gains in performance are obtained when the word-error-rates are higher.

A lot of the variance in task success in each of the two experimental conditions is therefore explained by the word-error-rate of each individual user. A more sensistive statistical analysis of the impact of the belief updating models on task success can therefore be performed if we also take into account this additional word-error-rate factor. We therefore performed a statistical analysis of covariance (ANCOVA) using task success (TS) as the independent variable, the experimental condition (Condition) as a main effect and the average word-error-rate (WER) as a covariate.

$$\text{TS} \leftarrow \text{WER} + \text{Condition}$$

The analysis was performed both at the session- and user-level. At the session level, the task success variable is binary; we therefore performed a logistic ANOVA. The resulting model was:

$$\log\left(\frac{P(TS=1)}{P(TS=0)}\right) = 2.085 - 0.049 \cdot \text{WER} + 0.681 \cdot \text{Condition}$$

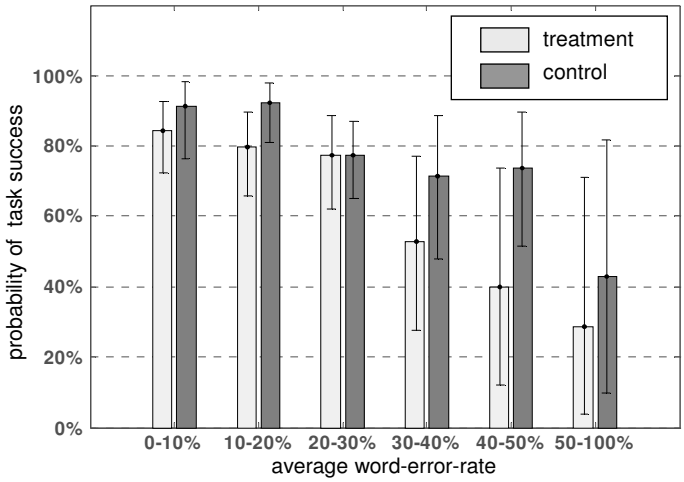


Figure 82. Empirical probability of task success at different WER for the treatment and control conditions

Adding the WER as a covariate in the model indeed improves the sensitivity of our statistical analysis and confirms that the experimental condition exerts a statistically significant impact on task success: $p=0.0102$. The effect of the WER is also significant, at a p-value $< 10^{-4}$. Based on this fitted model, we plotted the expected probability of task success as a function of WER in the two experimental conditions in Figure 83.A. The proposed models lead to gains in task success across a wide range of WER. As expected, at low word-error-rates the improvements are relatively small (see Figure 83.B.). As the word-error-rate increases, the expected improvement in the probability of task success also increases, and reaches a maximum of 16.3% absolute for a WER of 47%. Finally, at very high word-error-rates the size of the improvement decreases again. This profile can be explained by the fact that at low word-error-rates the recognition results are mostly correct, and simple heuristic belief updating rules will suffice. At the same time, very high word-error-rates limit the ability of any belief updating mechanism to construct accurate beliefs. In the extreme, if the correct value is never hypothesized by the recognizer, it will be nearly impossible to construct an accurate belief.

In the middle WER range, the proposed belief updating models allow the system to construct more accurate beliefs, which in turn lead to significant improvements in task success. For instance, at an average WER of 30%, the belief updating models produce a 14% absolute improvement in task success, from 64% to 78%. To attain the same task success with the control system, we would

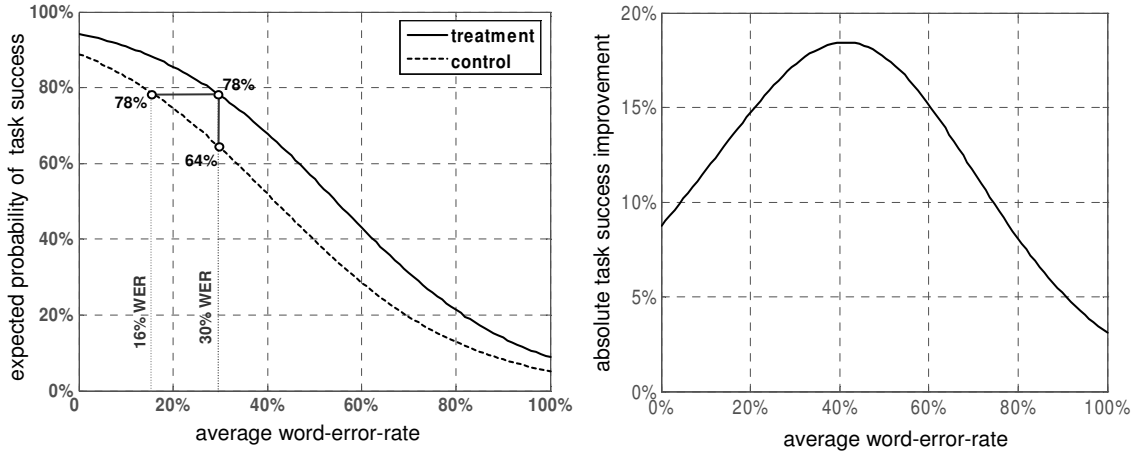


Figure 83. A. Expected session-level probability of task success as a function of word-error-rate in the treatment and control conditions; B. absolute improvement in task success at different word-error-rates

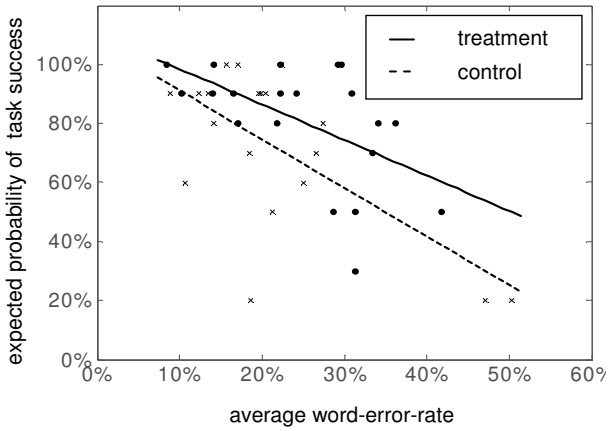


Figure 84. Expected user-level probability of task success as a function of word-error-rate in the control and treatment conditions

need a word-error-rate of 16%. In this case, the improvement is equivalent with cutting the word-error-rate in half. More specifically, the proposed models improve the log-odds of task success by the same amount as a 13.8% absolute reduction in WER (this number is obtained by dividing the corresponding weights for the WER and Condition in the ANCOVA model 0.681 / 0.049).

A second analysis of covariance was performed at the user-level. In this case the independent variable in the model was the average per-user task-success rate, the main effect was the experimental condition and the average per-user word-error rate was the covariate. The results are illustrated in Figure 84. Both the word-error-rate ($F=21.07$, $p=0.0001$) and the experimental condition ($F=4.44$, $p=0.0423$) exerted a statistically significant effect on the average per-user task success rate. The regression lines from Figure 84 again reflect that larger improvements are obtained within the user population with larger word-error-rates.

6.5.2.3 Impact on efficiency

In the previous subsection, we have seen that the proposed belief updating models significantly improve the probability of task success, across a wide range of word-error-rates. In addition, we also investigated the impact of the proposed models on the efficiency of the interaction. We suspected that, apart from leading from more successes, the increased system ability to construct accurate beliefs would also help users complete the tasks in a shorter period of time.

We performed an analysis of variance using task duration for successful tasks as the independent variable (TD). Task duration was measured as the number of turns to completion, and modeled as a Poisson variable. Like for the task success analysis, we used the experimental condition as the main effect and the session-average word-error rate as a covariate. Since the scenarios involved a different degree of complexity, they inherently had different mean durations. We normalized for the identity of the scenarios by introducing an offset variable in the model (this is equivalent to dividing each session length by the mean length of that particular scenario). The model therefore was:

$$TD \leftarrow Offset + WER + Condition$$

The resulting fitted model is described by the following equation

$$\log(TD) = -0.21 + 0.013 \cdot WER - 0.106 \cdot Condition + \log(offset)$$

Both the word-error-rate ($p<10^{-4}$) and the experimental condition ($p=0.0003$) exert a significant effect on task duration. This time, the effect of the word-error-rate is positive, i.e. the higher the word-error-rate, the longer the time to completion. The experimental condition exerts a negative impact on duration. The fitted normalized duration as a function of word-error-rate in the two experimental conditions is illustrated in Figure 85. Users (successfully) complete tasks faster with the sys-

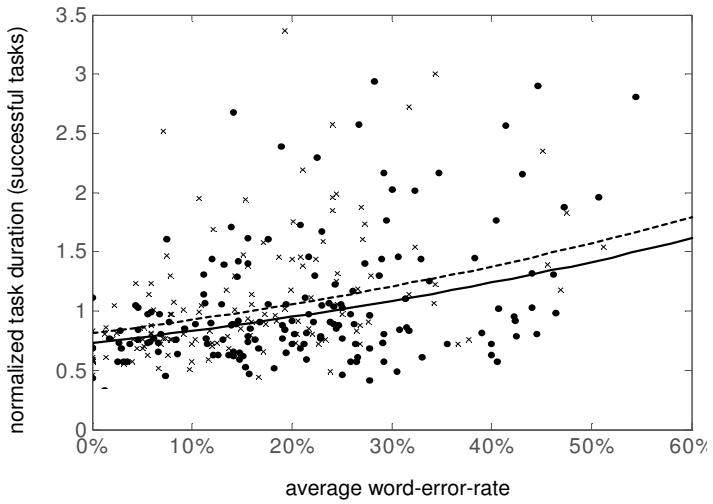


Figure 85. Expected task duration (# turns) for successful tasks as a function of word-error-rate in the control and treatment conditions

tem that uses the new belief updating models. The improvement produced by the belief updating models is equivalent in this case with a 7.9% absolute reduction in word-error-rate (0.106/0.013 = 7.9).

6.6 Summary and future directions

To date, various machine learning methods have been proposed for detecting misunderstandings [22, 44, 53, 70, 85, 102, 104, 117, 130] and corrections [54, 63, 68, 69, 121] in spoken dialog systems. Typically, these detectors work at the turn level (either on the whole utterance or on individual concepts in the utterance.) In this chapter, we have argued that spoken dialog systems should integrate evidence across multiple turns in a conversation and continuously monitor and improve the accuracy of their beliefs. We have proposed and evaluated a scalable, data-driven approach for this belief updating problem.

The approach relies on a compressed representation of beliefs and tracks up to k hypotheses for a concept at any given time (k is a small integer). The rest of the probability mass corresponding to all the other possibilities is clustered into a single "other" category. Using this representation, the belief updating problem can be cast as a multinomial regression task, and a data-driven approach becomes tractable. A multinomial generalized linear model can be trained from data to perform the updates. Experimental results discussed in this section show that the proposed approach significantly outperforms the heuristic rules currently used for this task in current systems. Furthermore, a user study with a mixed-initiative spoken dialog system shows that the approach leads to significant gains in task success and in the efficiency of the interaction, across a wide range of recognition error rates.

Independent of the gains in belief accuracy, and dialog effectiveness and efficiency, the proposed method has several other desirable properties. First, the proposed approach learns from data, tracks multiple hypotheses, and integrates information across multiple turns in the conversation. The idea of updating beliefs through time in spoken language interfaces also appears in previous work. For instance Higashinaka et al. [51] keep track of multiple dialog-states and resolve the ambiguities using empirical probabilities of dialog-act / dialog-act and dialog-act / dialog-state sequences. Before that, Paek and Horvitz [83] used a handcrafted Dynamic Bayesian Network to continuously update the belief over a user's goal in a command-and-control application. We take inspiration from their work and we automatically induce the models from data. One important advantage of the proposed machine learning technique (multinomial generalized linear models) is that it allows us to consider a very large number of features from different knowledge sources in the system, and it does not require expert knowledge about the potential relationships between these features. The most informative

features are automatically selected and weighted accordingly.

Second, the proposed approach is sample efficient and scalable. The focus is on a local, one-turn, rather than a global optimization. Furthermore, beliefs over each concept are updated independently: no inter-concept dependencies are modeled at this point. Such dependencies could however be taken into account in the proposed framework by adding inter-concept features. Although we sacrifice a theoretical notion of global optimality, learning is feasible even with relatively little training data. We have shown that this approach can lead to significant gains in task success and dialog efficiency in a real-world, fairly complex spoken dialog system. RoomLine operates over a space of 28 concepts with cardinalities ranging between two to several hundred possible values. Our results indicate that good local decisions can sum up to improvements in overall performance.

Third, the approach is portable. The belief updating framework was implemented as part of a generic dialog engine, decoupled from any particular dialog task. Modulo training data requirements, the approach can be reused in any new domain. More importantly, the approach does not make any strong assumptions about the dialog management technology used (e.g. form-filling, plan-based, task-oriented, information state update, etc). Consequently, the system developer is not tied to any particular type of dialog controller. The approach we have discussed in this chapter has been constructed in the context of a plan-based dialog manager, but is applicable in the context of any other dialog manager, as long as a notion of concept/slot/goal with multiple hypotheses exists.

The belief updating framework described in this chapter also raises a number of interesting follow-up questions. First, we believe that further performance improvements are possible. We have seen that high-level and domain-specific features play a very important role in these models. We believe that identifying and leveraging additional high-level knowledge such as domain-specific constraints or inter-concept dependencies can lead to additional performance gains. Furthermore, the models we have discussed so far have not made use of n-best list information. This was an artifact of the particular system we experimented with: our speech recognizer was not fast-enough to produce n-best lists online. The structure of the proposed belief updating models does however enable the use of n-best list information. In the presence of a speech recognizer that can generate a dense concept-level n-best list, further performance improvements might be possible.

The abstracted nature of the compressed belief representation also opens up a number of other opportunities in terms of model structure. For instance, the $\mathbf{BU}_{n=1}^{m=1}$ model discussed in subsection 6.4.5 remembers in each belief updating step the top initial hypotheses {h1} and adds one new hypothesis from the recognition result {r1}; the model constructs a new belief over the space {h1, r1, other}. The model does take into account confusability information, but only in the form of scores that reflect how confusable each of the h1 and r1 hypotheses is. Knowing for instance that the new value heard from the recognizer (r1) is in general confusable is useful, but knowing which value r1 is most often confused with can provide additional information; this latter information is not used by the current model. One way to take such information into account would be to construct a belief updating model with a slightly different structure. In each belief updating step, the model would remember the top initial hypothesis {h1} and would add one new value from the recognition result {r1}, but also the value that is most confusable with r1 – let's call it cr1. (Pre-computed confusability pairs would be necessary in this case.) The model would then construct an updated belief over the space {h1, r1, cr1, other}.

The belief updating models we have discussed in this chapter were trained via a supervised learning technique: multinomial regression models. The multinomial regression model is the natural extension of the logistic regression model to the case when the output variable is multinomial. This modeling technique has a number of good properties: the models are simple, can be trained from relatively little data and an automatic feature selection mechanism is integrated in the model building process (i.e. stepwise regression.) However, other multi-class supervised learning techniques could be used for the same task, such as decision trees, Bayesian networks, multi-class SVMs, etc. In the future, it would be interesting to comparatively evaluate several such techniques, like we did on the confidence annotation task in the previous chapter. Furthermore, in section 5.5 we have outlined two

general drawbacks of the supervised learning paradigm in the context of spoken language interfaces: (1) the require a pre-existing labeled corpus which is expensive and labor intensive to acquire, and (2) they favor "batch"-style training which does not match well the dynamic nature of the environments in which these systems operate. To address these drawbacks, in the previous chapter on confidence annotation we have investigated methods for generalizing error detection models across new domains, and we have proposed an implicitly-supervised learning paradigm for training models in an online fashion by extracting the supervision signal directly from the interaction. In the future, it would be interesting to investigate the same issues in the context of the proposed belief updating models: can we transfer or easily adapt models trained in one dialog domain to a different domain? Can we train these models online, by leveraging naturally-ocurring patterns in the interaction?

The belief updating models described in this chapter operate over concepts, i.e. they allow us to infer and update beliefs over the various pieces of information that the system acquires from the user. They do not provide any information about the correctness of the system state (i.e. are the system and the user grounded in terms of the current state in the task?) or about how to guide the system actions. In fact, in the RavenClaw arhictecture, there is no direct, explicit notion of state. Rather, the state is implicitly defined by the current values (and corresponding uncertainties) of the concepts acquired from the user and by the current dialog stack (see section 3.2.3.) Nevertheless, we believe that introducing a model of uncertainty and performing belief updates over the system state (in this case over the dialog stack) can provide an even more flexibility and can further increase the error handling abilities in the system. For instance, in [56], Horvitz and Paek articulate a conversational architecture for goal-oriented dialog that represents the possible user goals in a hierarchical fashion and allows for inference at different levels in this hierarchy. The authors show how value-of-information analyses can be used to guide the system's actions and to allow it to make principled decisions about when to backtrack in the dialog. In the future, it would be interesting to consider how a similar model of state uncertainty could be trained from data and integrated in a task-oriented dialog management framework like RavenClaw.

Finally, we believe the compressed belief calculus described in this chapter is applicable in a number of other problems beyond updating system beliefs in conversational spoken language interfaces. The "k+other" representation can be used in any problem that involves updating a large-degree multinomial distribution through time. Generally these are detection problems with large sets of possible outcomes. The "k+other" calculus can be seen as a useful heuristic that, under certain conditions makes learning-based approaches tractable. To fully exploit this heuristic, we need however to better understand its properties, advantages and limitations. A number of interesting theoretical questions arise. For instance, what is the relationship between a "k+other" multinomial model and the corresponding full multinomial model? Can we construct any guarantess (i.e. bounds) regarding the performance loss we can expect due to the lossy "k+other" compression? Intuitively, the loss depends on k, on the total number of possible outcomes, and on the nature of the underlying process that generates the observations throughout time. For instance, in the particular context of updating beliefs in spoken dialog systems, we have seen that, given the nature of the speech recognition process, the system hears (observes) only a limited set of possible outcomes throughout a conversation (i.e. throughout time). As a consequence, a small value for k is sufficient in this case, and we have empirically verified that further increasing k does not lead to any additional performance gains. This will however not be the case for any setting. In fact, adversarial conditions that lead to significant performance degradations can be easily envisioned. Clearly, k introduces a trade-off between tractability and expected- or perhaps worst-case performance. In the future, it would be interesting to start from a theoretical characterization of the process that generates the underlying observations, and, based on it, study these tradeoffs and create more principled solutions for choosing k.

PART IV.

HANDLING NON-UNDERSTANDINGS

Chapter 7

Rejection threshold optimization

We now turn our attention to the second type of understanding-error that affects spoken language interfaces: non-understandings. More precisely, in this chapter we address the issue of rejection non-understandings, and the trade-off it introduces between non-understandings (in the form of false-rejections) and misunder-standings. We start from the assumption that different understanding-errors have different costs at different points in the dialog. We develop a data-driven method that allows us to infer these costs by relating the number of errors to global dialog performance. Once the costs are known, we use them to optimize state-specific utterance rejection thresholds in a principled manner. We applied this method on data collected with the RoomLine systems. The resulting costs and state-specific rejection thresholds confirm our expectations, and are consistent with other anecdotal evidence gathered throughout the use of the system.

7.1 Introduction

Detection of non-understandings is, in general, an easy task. By definition a non-understanding is a situation in which the system knows the user took a turn (because a speech signal was identified), but fails to construct a valid discourse-level interpretation for the recognized result. In subsection 2.1.3 from Chapter 2 we have defined three types of non-understandings: **no-input**, in which no semantic interpretation can be obtained for the current recognition hypothesis; **unexpected-input**, in which a semantic representation is generated, but this representation cannot be incorporated in the current discourse structure; and **rejection**, in which the system deliberately rejects an input because of a low confidence score. The first two types, no-input and unexpected-input non-understandings can be detected automatically by definition. The last type, rejection non-understanding, does however raise an interesting question: when should a system reject an utterance? (i.e. how low does the confidence score have to be?)

The rejection mechanism is used as a device to guard against potential misunderstandings. We have seen in the previous chapters that spoken language interfaces typically use confidence scores to assess the reliability of the information contained in the recognition results. If an utterance has a very low confidence score, the system might decide that the likelihood of a misunderstanding is too high, and reject the utterance altogether. In effect, the system creates a rejection non-understanding in order to avoid a potential misunderstanding. Figure 86 provides a couple of examples. In the RavenClaw/Olympus infrastructure rejection is performed at the utterance level, i.e. a whole utter-

Example 1: (rejection non-understanding in turn 2 / true-rejection)

```
1  S: For when do you need the room?
2  U: next Thursday noon to two
      R: NEXT DAY ONE TO / 0.11
      P: [day=tomorrow, start_time=one]
```

Example 2: (rejection non-understanding in turn 2 / false-rejection)

```
1  S: How else can I help you today?
2  U: I need to make sure the room will hold forty people and has a network connection and a data projector
      R: I NEED TO RESERVE A ROOM FOR HOLD FORTY PEOPLE AND HAS NETWORK CONNECTION BENNETT DATA PROJECTOR / 0.14
      P: [size=40, equipment=network; projector]
```

Figure 86. Sample rejection non-understandings in a conference room reservation system; example 1 shows a true-rejection, example 2 shows a false-rejection
(`S:` marks the system turns, `U:` marks the user turns, `R:` marks the recognition result, `P:` marks the semantic representation of the recognition result)

ance is rejected at a time due to a low confidence score. This however need not be the case; rejection could also be performed on a concept-by-concept basis.

Spoken dialog systems typically decide to reject utterances by comparing the confidence score against a certain **rejection threshold**: if the score falls beneath this preset threshold the risk of misunderstanding is considered too high and the system rejects the utterance. Unfortunately, as we have already seen from the previous chapters, confidence scores are not perfectly accurate. Incorrectly understood utterances can have high confidence scores; conversely, correctly understood utterances can have low confidence scores. As a consequence, the system will sometimes reject correctly understood utterances; we deem these false-rejections. The use of a rejection mechanism[18] introduces a trade-off between the number of misunderstandings and false-rejections. As the rejection threshold increases the system becomes more conservative: it accepts only inputs that have high confidence, and as a result the number of misunderstandings will decrease. This happens however at the cost of increasing the number of false rejections. The corresponding trade-off curves are illustrated in Figure 87.A.

Another way to think about this trade-off is in terms of correctly and incorrectly transferred concepts. In each utterance, the user tries to convey one or more concepts to the system. If the confidence score is below the rejection threshold, the system rejects the utterance and no concept is transferred. On the other hand, if the system accepts the utterance, some concepts will be transferred correctly, while others might be misrecognized (in general, any given utterance might contain a mixture of correctly- and incorrectly-recognized concepts). Ideally, we would like to maximize the number of correctly transferred concepts and minimize the number of incorrectly transferred ones. However, as we raise the rejection threshold in order to lower the number of incorrectly transferred concepts, the number of correctly transferred ones also decreases, as shown in Figure 87.B.

The question we address in this chapter is: **given the existence of this trade-off, how can we set the rejection threshold in a principled manner?** We begin by discussing a number of current approaches for this problem in the next section; these approaches are heuristic in nature and do not take into account the fact that the optimal trade-off between non-understandings and misunderstandings might vary at different points in the dialog. Next, in section 7.3 we formalize the task at hand. Then, in section 7.4 we propose a principled, data-driven approach for inferring state-specific

[18] this rejection mechanism is commonly encountered in most spoken language interfaces. However, it is not mandatory: some systems might accept (and perhaps explicitly confirm) all inputs; as we shall see in this chapter, experimental results indicate that a "never-reject" strategy might be more appropriate in a number of situations.

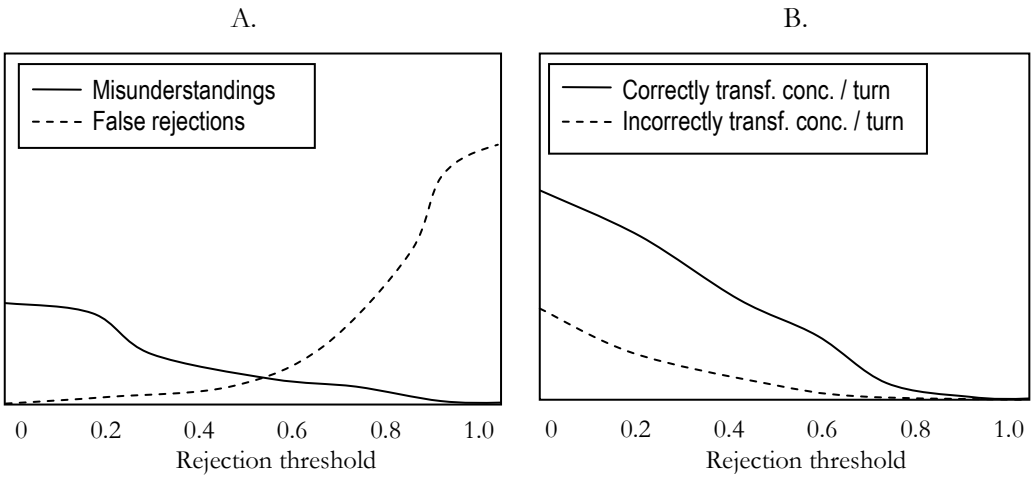


Figure 87. Typical rejection trade-off curves
A. misunderstandings and false rejections. B. correctly and incorrectly transferred concepts per turn

costs for various types of understanding-errors; we show how this technique can be used to optimize the non-understanding/misunderstanding trade-off in a state-specific manner. Finally, in section 7.5 we summarize the results and outline several potential directions for further extending this work.

7.2 Related work

Oftentimes rejection thresholds are set either at some arbitrary point (e.g. 0.3), or according to various rules-of-thumb and existing recommendations. For instance, the Nuance speech recognition engine [77] has a default value of 45 for the rejection threshold (the range is 0-100); the manual strongly recommends that this value is not to be changed – see Figure 88. Such an apriori, fixed rejection threshold is likely to be suboptimal for a number of reasons. Confidence annotators included in off-the-shelf speech recognition engines are typically optimized with respect to word-error-rate, while in the context of a spoken dialog system semantic-error is more important (see Chapter 5). Even if the annotator captures semantic confidence, intuitively the relative costs of false-rejections and misunderstandings are likely to vary across different systems. Moreover, these costs can vary even across different dialog states within the same system (some pieces of information are more important than others). As a consequence, the trade-offs between these types of errors, and hence the optimal rejection threshold is likely to be different in each of these conditions.

Another common approach is to empirically construct the trade-off curves and optimize the threshold based on rules-of-thumb regarding the costs of misunderstandings and rejections [31, 61, 91]. If the misunderstandings in a corpus are manually labeled, these trade-off curves can be inferred by counting how many misunderstandings and false rejections would occur in the system at various simulated rejection thresholds. Once the trade-off curve is available, a frequently used rule-of-thumb

Property	Value
nuance.rec.ConfidenceRejectionThreshold	0 - 100

Details: This property sets the confidence value that a user utterance must score in order to be considered a valid grammar match. When the value for a utterance is lower than this threshold, the system replaces the recognized result with an empty string. You can set this parameter to 0 to avoid any rejection based on the confidence score of a recognition result. Note that in all likelihood, you will never need to adjust this setting from its default value of 45.

Figure 88. Rejection threshold recommendation in Nuance speech recognizer documentation

is to assume that these costs are equal and therefore minimize the total sum of errors (i.e. find the break-even point.) Another commonly used rule-of-thumb is that misunderstandings are twice more costly than rejections; this rule is justified by the observation that accepting incorrect information is in general more costly since the user will have to spend one (or even more) future turn(s) on correcting this error.

In subsection 2.3.1 from Chapter 2 we introduced a data-driven model for assessing the impact of various types of understanding-errors on global dialog performance. We showed that this type of model can allow us to infer the relative costs of misunderstandings and non-understandings, with respect to a desired global performance metric. In this chapter, we start from the intuition that the costs for different types of understanding-errors might vary across domains and even across different dialog states. We expand on the models described in Chapter 2 by introducing distinctions between different dialog states. The new models allow us to capture the costs of various types of understanding-errors at different points in the dialog and to optimize the rejection threshold in a principled, state-specific manner.

7.3 Problem statement

The problem we are addressing can be stated as follows:

> How can we compute in a principled manner the relative costs of misunderstandings and false rejections at different points in the dialog? Given these costs, can we adjust the rejection thresholds in a state-specific fashion, such as to maximize a chosen global performance metric?

7.4 Data-driven error-cost assessment and rejection threshold optimization

We propose a data-driven approach for this problem. We describe the proposed method in the next subsection, 7.4.1. Then, in subsection 7.4.2 we present a set of empirical results obtained by applying the proposed method in the RoomLine system. In the next section, 7.5, we present a number of concluding remarks and directions for further extending this work.

7.4.1 Method

7.4.1.1 A data-driven model for error-cost assessment

We have already introduced a method for assessing the impact of various types of understanding-errors on global dialog performance in our analysis from subsection 2.3.1, Chapter 2. We begin by recapping this method. Then, in subsection 7.4.1.2 , we extend it to infer state-specific error-costs.

The central idea behind the proposed error-cost assessment method was to construct a regression model that relates the number (or proportions) of various types of understanding-errors in a given dialog to the overall performance in that particular session. We illustrate again the proposed method using an example. Assume we are interested in optimally balancing the average number of correctly and incorrectly transferred concepts per turn. Let's call these variables CTC and ITC. Note that these quantities vary with the rejection threshold: as the threshold increases, the average number of incorrectly transferred concepts per turn will decrease, but so will the average number of correctly transferred concepts – see Figure 87.B. In addition, let's assume we want to optimize for task success (TS) modeled as a simple binary variable. We can determine the relative costs of correctly and incorrectly transferred concepts with respect to task success by fitting a logistic regression model [76] using CTC and ITC as predictor variables, and designating TS as the dependent variable. Each datapoint in the regression model corresponds to an entire dialog session. Now, let's assume we obtain a good fit for the model, represented by the following regression equation (this is a fictitious example):

$$\log\left(\frac{P(TS=1)}{P(TS=0)}\right) = 0.21 + 2.14 \cdot CTC - 4.12 \cdot ITC$$

This equation tells us that on average an incorrectly transferred concept has a cost (-4.12) about twice as high as the utility (+2.14) a correctly transferred concept. With these costs and the trade-off curves for CTC and ITC in hand, we can find the threshold that maximizes the overall utility: we compute, plot and find the maximum point for the curve 2.14·CTC – 4.12·ITC. In using a rejection threshold that maximizes this expression, we are implicitly maximizing the log odds of task success, according to the constructed regression model.

The proposed method can be summarized in four steps:

(1) identify a set of variables A, B, … involved in the rejection tradeoff; these parameters vary with the rejection threshold (th):

$$A = A(th)$$
$$B = B(th)$$

(2) choose a global dialog performance metric P to optimize for;

(3) fit a model m that relates the trade-off variables to the chosen global dialog performance metric:

$$P \leftarrow m(A, B)$$

(4) find the rejection threshold that maximizes the performance:

$$th^* = \arg\max_{th}(P) = \arg\max_{th}(m(A(th), B(th)))$$

In general, the performance metric P (i.e. the dependent variable in the regression model) can be any objective or subjective global dialog performance metric. Some candidates include: task success (measured either as a binary variable or using the Kappa agreement statistic on an attribute-value matrix in a slot filling system [128]), task duration, user satisfaction, etc. Objective metrics such as task success and duration are better suited for this type of modeling. Subjective metrics exhibit large variance not only due to the system's performance, but also due to variations in user's expectations [48]. As a consequence, it is generally more difficult to build predictive models using these metrics, unless sufficiently large amounts of data are available. The nature of the performance metric P dictates the type of the regression model. For instance, if we are interested in optimizing for binary task success, the logistic regression model is the most appropriate – the output variable is distributed according to a binomial distribution. On the other hand, task duration, expressed at the total number of turns to completion, is Poisson distributed; in this case a Poisson generalized linear model would be more appropriate [76].

Any variables affected by the rejection trade-off can be used as predictors in the model. In the example discussed above, we used the average number of correctly and incorrectly transferred concepts per turn (CTC and ITC). In general, any variables that are involved in the rejection trade-off could be used, such as the relative proportions of misunderstandings and false-rejections in a conversation.

7.4.1.2 A data-driven model for state-specific error cost assessment

The approach discussed above allows us to assess the relative costs of different understanding-errors with respect to a global dialog performance metric. This assessment is however global in nature. In the first section of this chapter, we have argued that different errors might have different costs at different points in the dialog. In this subsection we extend this methodology to determine state-specific error costs, and therefore find state-specific rejection thresholds.

The central idea is to use a different, more refined set of predictor variables, defined on a

state-by-state basis. For instance, instead of using the average number of correctly and incorrectly transferred concepts per turn in a session, we can define the same variables on a per-state basis. Assume a system operates with n dialog states numbered 1 to n. The regression model then becomes:

$$P \leftarrow m(CTC_1, ITC_1, CTC_2, ITC_2, ..., CTC_n, ITC_n)$$

where CTC_i and ITC_i capture the average number of correctly and incorrectly transferred concepts per turn in state i. The coefficients resulting from the regression model reflect the impact, and therefore the cost of a correctly or incorrectly transferred concept on the chosen dialog performance metric (P). These costs can be used to optimize separate rejection thresholds for each of the states under consideration.

The proposed approach suffers from one important limitation. In general, only a relatively small number of states can be taken into consideration. The number of predictor variables in the regression model equals the product of the number of variables involved in the rejection trade-off and the number of states considered. Unless large amounts of training data are available, it will be impossible to build a robust regression model using a correspondingly large set of predictor variables. A second assumption made by this model is that changing the rejection threshold in one state does not affect the rejection trade-off curve for a different state; in other words, we treat the rejection mechanisms for each state independently.

In systems where the number of underlying states is large compared to the amounts of data available for building the error-cost assessment model, the scalability issue can be circumvented by clustering the states into several groups, and determining costs and optimizing thresholds for each state-cluster rather than for each state. State clustering can be performed heuristically, by a system developer, based on domain-specific knowledge. Alternatively, data-driven methods for clustering the states could be investigated. In the empirical work described in the following section we used the first method; we briefly comment on a potential data-driven approach to clustering in the concluding remarks for this chapter.

The proposed method can be formalized as follows:

(1) identify a set of variables A, B, … involved in the rejection tradeoff

(2) define the set of states, or state clusters $\{s_i\}_{i=1..n}$

(3) choose a global dialog performance metric P to optimize for;

(4) fit a model m that relates the trade-off variables (computed on a per-state or per-state-cluster basis) to the chosen global dialog performance metric:

$$P \leftarrow m(A_1, B_1, A_2, B_2, ..., A_n, B_n)$$

(5) for each state or state-cluster, find the optimal threshold that maximizes performance:

$$th^* = \arg\max_{th}(P) = \arg\max_{th}(m(A_1(th), B_1(th), A_2(th), B_2(th), ..., A_n(th), B_n(th)))$$

7.4.2 Experimental results in the RoomLine domain

Next, we describe results obtained using the proposed state-specific error cost assessment methodology in the RoomLine domain.

7.4.2.1 System and data

The experiments were conducted in the context of RoomLine, a mixed-initiative spoken dialog system for conference room reservations. The corpus used in these experiments was collected through a user-study in which 46 participants performed each up to 10 scenario-based interactions with the system. Each scenario required the user to make a room reservation, within a specified set of con-

straints. The RoomLine system was described extensively earlier, in subsection 3.4.1 from Chapter 3. The data collection experiment is described extensively later, in subsection 8.3.1 from Chapter 8.

Here, it suffices to say that the corpus contained 449 dialog sessions and 8278 user turns. The user speech data was orthographically transcribed by a human annotator, and checked by a second annotator. Each dialog session was labeled as successful or not, depending on whether or not the user completed the scenario as instructed. Based on an automatic comparative analysis of the decoded results and reference transcripts, each user turn was annotated with the number of correctly and incorrectly transferred concepts. If a turn contained at least one incorrectly transferred concept, it was labeled as a misunderstanding.

Throughout the data collection experiment, the system used a global, fixed rejection threshold of 0.3.

7.4.2.2 State clustering

The state-space for the RoomLine system subsumes 71 states. Since we cannot reliably build a regression model with 142=71x2 predictor variables using just 449 data-points (each session is a data-point in the model), we resorted to the state clustering methodology we described earlier in subsection 7.4.1.2. We manually clustered the 71 states into 3 classes which we suspected would exhibit different characteristics in terms of the rejection trade-off. The first state cluster, `open-request` (or S1), included the states in which the system asked an open question such as "How may I help you?" The second state cluster, `request(bool)` (or S2), included the states in which the system requested a Boolean concept, i.e. a yes/no answer from the user (e.g. "Do you want a reservation for this room?"). Finally, the last state-cluster, `request(non-bool)` (or S3), included all the other states, in which the system requested a concept with more than 2 possible values from the user (e.g. "Starting at what time do you need the room?"). We believe that finer distinctions could be made if larger amounts of training data were available.

7.4.2.3 Optimizing for task success

In a first set of experiments, task success was used as the target for optimization. The predictor variables were the average number of correctly and incorrectly transferred concepts per turn, for each of the three dialog state-clusters discussed above.

Given that task success is modeled as a binary variable, a logistic regression model was used. The model showed a good fit, increasing the average log-likelihood on the training set from a majority baseline of -0.4655 to -0.2927 ($p<10^{-4}$ in a likelihood ratio test). To confirm the robustness of the model and check for over-fitting, we also performed a 10-fold cross-validation procedure. We split the training set into 10 chunks and repeatedly trained a model on 9 of these subsets and tested on the remaining validation set. The average log-likelihood in this cross-validation process was 0.3136, close to the training set average log-likelihood, indicating a robust fit. In a hard metric evaluation, the regression model is able to predict task success with an error rate of 11.91% (the majority baseline was at 17.62%). The resulting model was:

$$\begin{aligned}\log\left(\frac{P(TS=1)}{P(TS=0)}\right) = & -2.40 \\ & +0.58\cdot CTC_{S1} - 0.37\cdot ITC_{S1} \\ & +3.39\cdot CTC_{S2} - 0.62\cdot ITC_{S2} \\ & +2.53\cdot CTC_{S3} - 3.56\cdot ITC_{S3}\end{aligned}$$

where CTC_{S1} and ITC_{S1} are the average numbers of correctly and incorrectly transferred concepts per turn in state-cluster S1 (`open-request`), CTC_{S2} and ITC_{S2} are the average numbers of correctly and incorrectly transferred concepts per turn in state-cluster S2 (`request(bool)`), and CTC_{S3} and ITC_{S3} are the average numbers of correctly and incorrectly transferred concepts per turn in state-cluster S3 (`request(non-bool)`).

The regression coefficients are shown in Table 35, together with their corresponding standard errors and p-values (the null hypothesis is that the coefficient is 0). The regression coefficients reflect the impact of correctly and incorrectly acquired concepts on the probability of task success. As expected, the coefficients for correctly transferred concepts are all positive, while the coefficients for incorrectly transferred concepts are all negative. This result is in line with our prior intuition: a large number of correctly transferred concepts per turn increases the probability of task success, while a large number of incorrectly transferred concepts per turn decreases the probability of task success. Note also that the ratio of the costs for an incorrectly transferred concept and the utility for a correctly transferred is different in each state: in the `open-request` state, the ratio is 0.37:0.58 or 1:1.57; in the `request(bool)` state the ratio is 0.62:3.39, or 1:5.47; finally, in the `request(non-bool)` state the ratio is 3.56:2.53 or 1:0.71. These differences confirm the initial intuition that the average costs for various types of errors are different at different points in the dialog.

Next, we used the costs obtained via the regression model to find optimal thresholds for each of the three dialog-state-clusters, in light of the CTC/ITC trade-off curves. The CTC/ITC curves were computed empirically for each dialog-state-cluster, by simulating different rejection thresholds. We then computed the overall cost (or utility) as a function of the rejection threshold in each state-cluster. The CTC/ITC and utility curves are illustrated in Figure 89. The optimal threshold is the one that maximizes the corresponding utility function. For the `open-request` state-cluster the maximum utility is attained when the rejection threshold is at zero; or model indicates that in this state the system should use an always-accept (or never-reject) policy. Similarly, the optimal threshold for the `request(bool)` state-cluster is also zero. If for the `open-request` state-cluster, 0 is clearly the maximizing point, for the `request(bool)` state-cluster the utility profile has a large plateau indicating that for a wide range of threshold values (0 – 0.6), the utility stays roughly constant. Finally, in the `request(non-bool)` state-cluster, the utility function has again a clear maximum; that maximum is reached for a rejection threshold value of 0.61.

Because the standard errors on the regression coefficients (i.e. costs) raised some concerns, we performed an additional robustness check. We split the data into two halves, built separate models on each half and compared the results. The variations in the coefficients were minor and the resulting utility profiles had similar shapes for all states across the models; the optimal thresholds were at the same locations.

The resulting threshold values are consistent with anecdotal evidence gathered throughout the data collection experiment, and corroborate our prior intuitions. For instance, we noticed that long utterances (which were very frequent after the initial, open "How may I help you?" prompt) would generally have low confidence scores and would therefore be rejected by the system even though they were correctly recognized. This observation was confirmed by a later analysis that showed that in the `open-request` state the proportion of falsely-rejected utterances was much larger than in the other two state-clusters (17.4% as opposed to 2.0% and 1.7%). We conjecture that this behavior was caused by a mismatch between the data encountered by the system throughout the experiment and the data used to train the confidence annotator.

Variable	Coefficient	S.E.	p-value
Const	-2.40	1.16	0.0386
CTC / open-req	0.58	0.30	0.0510
ITC / open-req	-0.37	0.47	0.4286
CTC / req(bool)	3.39	1.01	0.0008
ITC / req(bool)	-0.62	1.32	0.6363
CTC / req(non-bool)	2.53	0.82	0.0019
ITC / req(non-bool)	-3.56	1.12	0.0014

Table 35. Regression coefficients (i.e. costs) for task success model

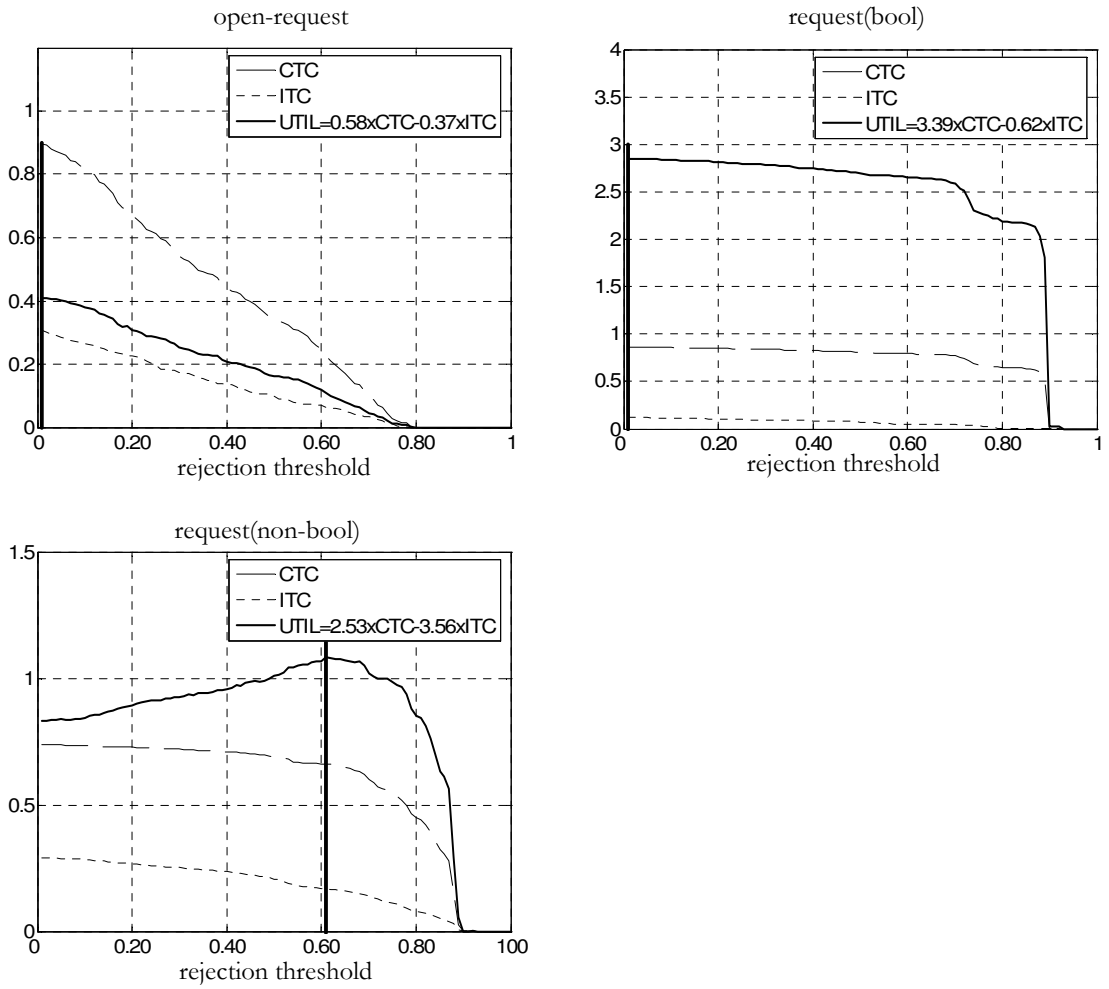


Figure 89. State-specific CTC/ITC tradeoff, utility, and rejection threshold optimization (for task success)

The data used to train the confidence annotation model consisted of calls to an earlier live version of the system. In this data a number of users were exploring the boundaries of the system, and long utterances were generally out-of-domain or out-of-application-scope and led to misunderstandings. This pattern was picked up by the confidence annotator, which generally assigned low confidence scores to long utterances. However, this pattern was no longer valid throughout the subsequent data collection experiment: the users were more goal-oriented and long utterances were generally in-domain and correctly understood. The data-driven error cost assessment model we developed indicates that the cost for incorrect concepts is not very high in this state (we are at the beginning of the dialog), relative to the utility for correctly transferred concepts: -0.40 versus 0.55. Taking also into account the potentially larger number of correctly transferred concepts in this state-cluster (due to longer utterances), the model correctly indicates that the system should use a threshold of zero, i.e. accept all utterances regardless of the confidence score. We believe this result shows how the proposed data-driven error cost model can indeed mitigate potential mismatches between the given confidence annotator and the characteristics of the domain in which the system operates. The approach is therefore well-suited to situations when a pre-trained, off-the-shelf confidence annotation model is used.

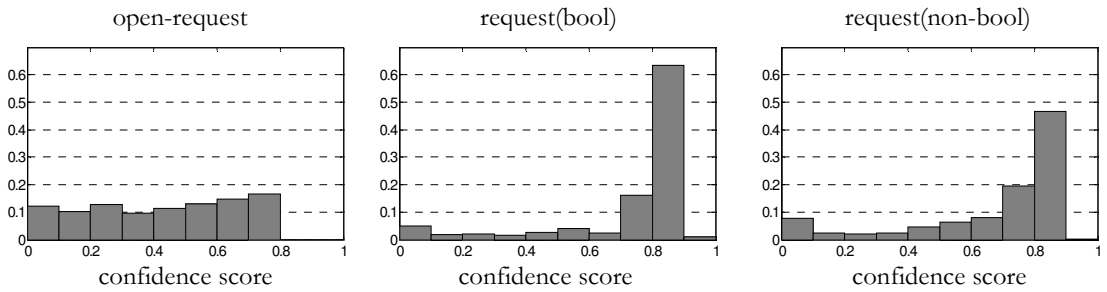


Figure 90. Confidence score histograms in the 3 state-clusters: open-request, request(bool) and request(non-bool)

We believe that the explanation for the large plateau in the resulting cost profile for the `request(bool)` state-cluster lies in the fact that most responses in this case are simple yes/no answers. These responses are typically easily recognized by the system, and are generally assigned high confidence scores (see Figure 90). As a result, for a wide range of low rejection thresholds, very few utterances are rejected, and the numbers of correctly and incorrectly transferred concepts, as well as the overall cost, stay roughly constant.

This is no longer the case in the `request(non-bool)` state-cluster. The distribution of confidence scores is less sharp in this state (see Figure 90), and as a result the utility changes more significantly as the rejection threshold varies. This is the only state-cluster where the cost of an incorrectly transferred concept (-3.44) exceeds in absolute value the utility of a correctly transferred concept (+2.55). This relationship is in line with the intuition that non-Boolean concepts are the most crucial ones for task success in this domain (e.g. the date, start-time, and end-time for the reservation as well as various characteristics such as room size and equipment). As a result, the system should be more conservative in this state, and accept only high-confidence responses: the optimal threshold is at 0.61.

The last question we investigated was: what changes should we expect in the system as we move the rejection threshold from its previous global setting at 0.3 to the new, state-specific values? Because dialog is an interactive process, another end-to-end experiment would be required to answer this question: changing the thresholds might in fact change the dynamics of the interaction, which in turn might lead to changes in the task success model. We can however construct an informed estimate by analyzing the effect of the new rejection thresholds on the average CTC/ITC numbers in each state-cluster; we can then use the current task success model to project the influence of the new CTC/ITC numbers on task success. The results are shown in Table 36. For the `open-request` state, we expect that the change in the rejection threshold will lead to an average increase of 0.35 correctly transferred concepts per turn, at the expense of a 0.15 increase in incorrectly transferred ones. For the `request(bool)` state-cluster the situation stays roughly the same as before (the CTC/ITC profiles are roughly constant in the 0.3-0.6 threshold range). For the `request(non-bool)` state-cluster we expect a small decrease in both the incorrectly and the correctly transferred concepts per turn. Finally, we can also project an expected value for the overall task success rate, by using the constructed logistic regression model and the new expected CTC/ITC values in each state-cluster. The

State	Variable	Current	New	Delta
open request	CTC	0.54	0.89	+0.35
	ITC	0.16	0.31	+0.15
request bool	CTC	0.84	0.86	+0.02
	ITC	0.09	0.12	+0.03
request non-bool	CTC	0.72	0.66	-0.06
	ITC	0.25	0.17	-0.08
Task success rate		82.75%	87.16%	+4.41%

Table 36. Estimated changes in CTC, ITC and task success

model predicts a 4.4% absolute increase in task success rate.

7.4.2.4 Optimizing for task duration

The methodology we have described in section 7.4.1 can be used to determine error-costs relative to any global dialog performance metric. In the previous subsection, we used task success as the target for optimization. In this subsection, we report experiments using a different optimization target: task duration for successful tasks.

Because task duration is expressed as the number of turns, we modeled it as a Poisson response variable in a generalized linear model [76]. Since the data consists of scenario-driven interactions with the system, we normalized for the inherent differences in durations between the 10 different scenarios in this corpus by introducing the mean duration for each scenario as an offset variable in the regression model. The resulting model was:

$$\begin{aligned}\log\left(\frac{D}{MSD}\right) = & -1.24 \\ & -0.16\cdot CTC_{S1} - 0.09\cdot ITC_{S1} \\ & -0.76\cdot CTC_{S2} - 0.36\cdot ITC_{S2} \\ & -0.54\cdot CTC_{S3} + 0.50\cdot ITC_{S3}\end{aligned}$$

where D is the task duration (expressed as number of turns) and MSD is the mean task duration for the corresponding scenario (log(MSD) is used as an offset variable in the regression model). CTC_{S1}, ITC_{S1}, CTC_{S2}, ITC_{S2}, CTC_{S3} and ITC_{S3} are again the numbers of correctly and incorrectly transferred concepts per turn in each state-cluster. The model showed a good fit. The correlation coefficient between the actual and predicted task durations is R=0.62, and is illustrated in Figure 91.

The regression coefficients are also shown in Table 37, together with their corresponding standard errors and p-values (the null hypothesis is that the coefficient is 0). The regression coefficients reflect the impact of correctly and incorrectly acquired concepts on normalized task duration. As expected, the coefficients for incorrectly transferred concepts are larger (more positive) in this case than the coefficients for correctly transferred concepts: they lead to increases in task duration. The ratio for the costs for an incorrectly transferred concept and the utility for a correctly transferred is again different across the three state-clusters.

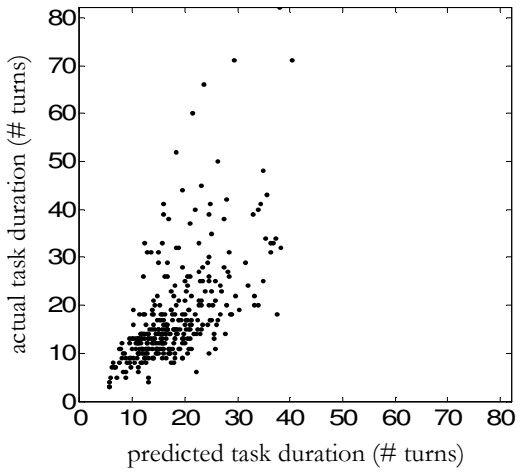


Figure 91. Correlation between predicted and expected task duration according to trained error cost model

Variable	Coefficient	S.E.	p-value
Const	1.2403	0.11	0.0000
CTC / open-req	-0.1650	0.02	0.0000
ITC / open-req	-0.0944	0.04	0.0276
CTC / req(bool)	-0.7630	0.09	0.0000
ITC / req(bool)	-0.3641	0.15	0.0141
CTC / req(non-bool)	-0.5440	0.05	0.0000
ITC / req(non-bool)	0.4979	0.10	0.0000

Table 37. Regression coefficients (i.e. costs) for normalized task duration models

We used the costs obtained in regression to identify the optimal rejection thresholds, this time with respect to minimizing task duration. The resulting utility profiles and optimal thresholds are illustrated in Figure 92. For the `open-request` and `request(bool)` state-clusters, the profiles are very similar to those obtained when optimizing for task success (shown in Figure 89), indicating again an optimal rejection threshold of 0 for these states. For the `request(non-bool)` state-cluster, the optimal threshold was again 0.61. However, in this case the utility profile had a less pronounced maximum: for a relatively wide range of threshold values (i.e. 0.2 – 0.6), the overall utility in this state-cluster stays roughly the same.

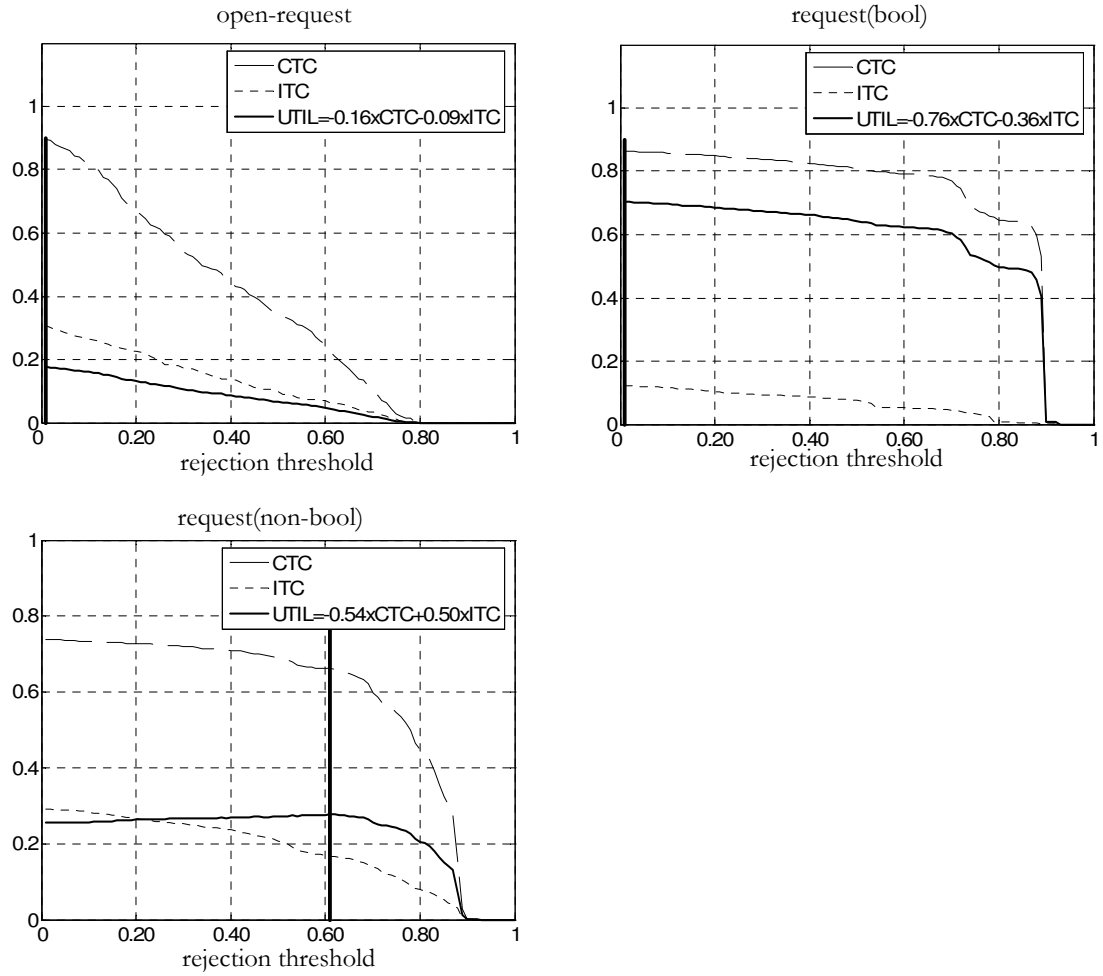


Figure 92. State-specific CTC/ITC tradeoff, utility, and rejection threshold optimization (for task duration)

7.5 Summary and future directions

Spoken dialog systems typically rely on confidence scores to guard against potential misunderstandings. If the confidence score for the current input is too low, the system might decide to reject the input altogether. In this case, a rejection non-understanding is purposefully created in order to avoid a potential misunderstanding. An inherent trade-off therefore exists between misunderstandings and non-understandings. The system can exert control over the proportions of non-understandings and misunderstandings in a dialog by adjusting the rejection threshold.

In this chapter we have proposed a principled method for optimizing rejection thresholds. The proposed method builds on top of a data-driven approach for inferring the costs for various types of understanding-errors in a spoken dialog system (the method was already introduced in Chapter 2.) The central idea behind this error-cost assessment method was to use a regression model to capture the relationship between the frequency of errors and global dialog performance. Once the model is fit, the regression coefficients reflect the cost of errors with respect to the chosen performance metric. The computed costs can then be used to optimize utterance-level rejection thresholds in a principled manner. We have started from the intuition that different errors have different costs at different points in the dialog and have shown that the proposed error-cost assessment methodology can be extended to derive these state-specific costs. In turn, these costs allow us to perform a state-specific optimization of rejection thresholds. Experiments conducted based on a dataset collected with the RoomLine system confirmed our expectations. The relative trade-offs between different types of understanding-errors are indeed different at different points in the dialog. The resulting optimal rejection thresholds are consistent with our intuitions and with other evidence gathered throughout the data collection experiments.

The proposed approach has a number of advantages over current solutions for setting rejection thresholds. First, the approach is data-driven. Instead of setting a rejection threshold based on rules-of-thumb or postulated costs, the error costs and therefore the thresholds are inferred from data. They are adapted to the particular characteristics of the domain and components (i.e. speech recognizer, confidence annotation model) with which the system operates. In fact in subsection 7.4.2.3 we have that the proposed approach bridges a mismatch between the distribution of the data encountered by the system at runtime and an off-the-shelf confidence annotator trained on different data.

Second, any chosen global dialog performance metric can be targeted for optimization. In this chapter we have experimented with task success and task duration; the resulting optimal thresholds were similar. The similarity is not surprising in this case. The RoomLine system operates in an information access domain, where the two metrics are related: "successful" and "short" go generally hand in hand in these domains. However, this relationship does not necessarily hold across all domains and interaction-types. For instance in a tutoring system, learning gain might be a more appropriate target for optimization than time-on-task.

Third, the proposed methodology can be used to infer state-specific costs for errors and state-specific rejection thresholds. In subsection 7.4.1.2, we have pointed out a limitation of this approach: the number of state distinctions that can be made is bounded by the amount of available training data. In our experiments, a three-way distinction was used – we manually identified three state-clusters that we believed exhibited similar properties in terms of the rejection trade-off. In future work, it would be interesting to investigate data-driven approaches to this state clustering problem. For instance, could we automatically identify the state clusters in which the error costs are similar? A potential method would be to start with a single initial cluster that contains all the states and iteratively split based on a criterion that reflects the goodness-of-fit of the resulting regression model. This approach would help ensure that number of resulting state-clusters is adequately balanced for the amount of training data available.

Finally, before we move on to the next chapter, we would like to bring forward another potential use of the proposed error cost assessment methodology. In this work, we have used it to balance the costs for different variables involved in the rejection trade-off. We believe the same ap-

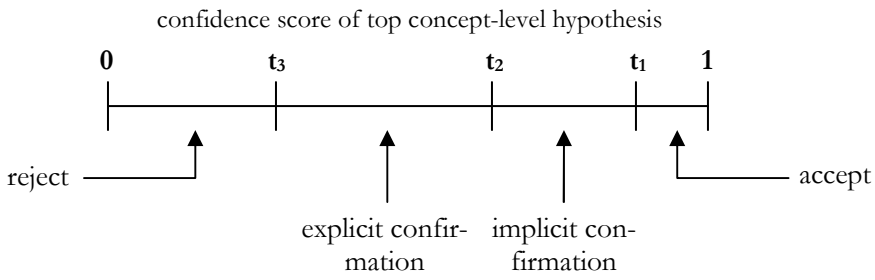


Figure 93. A typical misunderstanding recovery policy

proach could be used to optimize confidence thresholds for explicit and implicit confirmations and in effect build policies for recovering from misunderstandings.

Insofar, we have focused on whole-utterance rejection as a mechanism to guard against potential misunderstandings. This was an infrastructure constraint rather than a limitation in the proposed methodology. However, spoken dialog systems might use rejection at the concept-level (i.e. reject individual concept values rather than an entire utterance.) Other strategies, such as explicitly or implicitly confirming the current concept hypothesis may also be used to guard against potential misunderstandings. The decision to engage one of these actions is typically based on the confidence score of the current concept hypothesis. A typical policy is illustrated in Figure 93: reject the value if its confidence score is below t_3, explicitly confirm if the confidence score is between t_3 and t_2, implicitly confirm if the score is between t_2 and t_1, and accept that value as grounded if the confidence score is above t_1. We believe the technique described in this chapter could be used to optimize these thresholds (i.e. t_1, t_2, t_3) and therefore offers a principled solution to the problem of developing misunderstanding recovery policies.

Consider for instance a system that engages in four different actions to guard against misunderstandings: reject (R), explicit confirmation (EC), implicit confirmation (IC) and accept (A), as illustrated in Figure 93. To infer the three thresholds confidence, we need to estimate the costs incurred for engaging in each of these actions in two cases: when the concept value is correct, and when the concept value is incorrect. Let's denote by R_c the number of false-rejections (i.e. the rejected concept value was in fact **c**orrect in a dialog session) and by R_i the number of true-rejections (i.e. the rejected concept value was **i**ncorrect). Similarly, EC_c, IC_c, and A_c are the numbers of explicit confirmations, implicit confirmations and accepts with correct concept values; EC_i, IC_i, and A_i are the numbers of explicit confirmations, implicit confirmations and accepts with incorrect concept values. Like before, we can build a regression model that relates these variables to an overall dialog performance metric:

$$P \leftarrow m(R_C, R_i, EC_C, EC_i, IC_C, IC_i, A_C, A_i)$$

Once the model is fitted, the regression coefficients will reflect the costs for each of these actions. The corresponding optimal thresholds can then be easily inferred (see for instance Figure 39 from subsection 4.3.3 in Chapter 4.) If sufficient amounts of training data are available, further distinctions between different dialog states (in this case concepts) could be introduced in the model.

Chapter 8

Non-understanding recovery: strategies and policies

In this chapter we turn our attention to the problem of recovering from non-understandings. In the first part of the chapter, we describe an in-depth empirical analysis of ten non-understanding recovery strategies and two different recovery policies. The questions under investigation are: (1) how do various strategies for recovering from non-understandings compare to each other (both in terms of local recovery performance and subsequent user behaviors), and (2) can a good recovery policy lead to significant improvements in performance, both locally and globally? In the second part of the chapter, we propose a scalable, online, data-driven approach for learning non-understanding recovery policies over large sets of strategies. An evaluation performed in the context of a deployed spoken dialog system shows that the proposed approach leads to statistically significant improvements in the non-understanding recovery rate.

8.1 Introduction

We have argued that three components are required in order to successfully recover from understanding-errors: first, spoken language interfaces must be able to detect these errors, preferably as soon as they happen. Second, they must be equipped with a rich repertoire of error recovery strategies that can be used to set the conversation back on track (recall that by strategy we denote a simple, one-turn action the system might engage in to recover.) Third, systems need to know how to choose optimally between different recovery strategies at runtime; in other words, they must have good recovery policies.

We have seen that in the case of non-understandings, detection is in general a simple task (except for the special case of rejections, discussed in detail in the previous chapter.) On the other hand, developing good strategies and policies for recovering from non-understanding errors poses a number of significant challenges.

The number of strategies that could be used to recover from a non-understanding is relatively large. For instance, the system could ask the user to repeat; it could ask the user to rephrase; it could notify the user that an error has occurred; it could repeat the previous prompt; it could ignore the non-understanding and try to advance the task in a different manner; it could provide various

types of help messages, e.g. tell the user what she can say at this point, tell the user to speak softer or louder, or tell the user to move to a quieter place. And this is not an exhaustive list.

Intuitively, each of these strategies will be more appropriate under certain circumstances. For example, asking the user to repeat is not a good course of action if the non-understanding was the result of an out-of-grammar utterance. In contrast, if the non-understanding was caused by a transient noise, such as a door slam, asking the user to repeat is probably more likely to succeed. However, if detecting non-understandings is an easy task, diagnosing the actual source of the error is a challenging one. In addition, the relative trade-offs between non-understanding recovery strategies are not well understood; oftentimes, these trade-offs are task and system-specific. As a consequence, it becomes difficult to design provably good policies for choosing between different recovery strategies at runtime.

In this chapter, we focus our attention on issues related to non-understanding recovery strategies and policies. More specifically, we address the following questions:

- **How do various strategies for recovering from non-understandings compare to each other?** We believe that a better understanding of the various non-understanding recovery strategies and of user behaviors following these strategies can help improve their design and can provide useful insights for the developing good recovery policies.
- **Can a good non-understanding recovery policy lead to significant improvements in performance?** While the intuitive answer to this question is yes, we believe that an empirical validation of this hypothesis is necessary before we decide to focus efforts on developing recovery policies. The performance of the error recovery process is a product of both the set of available strategies and the policy used to engage them. If the set of strategies does not provide good coverage for the types of problems encountered by the system, it will be very difficult to significantly improve performance just by designing a new recovery policy. Should this be the case, our efforts would probably be better focused on developing more efficient recovery strategies, rather than on trying to learn a policy.
- **How can we develop good non-understanding recovery policies operating with large sets of recovery strategies?** We argued that, especially when the underlying set of non-understanding recovery strategies is large, it is difficult to manually construct heuristic policies. Ideally, we would like systems to learn recovery policies online, through trial-and-error, from their own experiences.

In an effort to address the first two questions, we performed an empirical analysis of ten non-understanding recovery strategies using a dialog corpus collected with RoomLine, a conference room reservation system. We introduced a number of local metrics for measuring recovery performance, and we used them to compare the ten non-understanding recovery strategies. Additionally, we analyzed the relationship between each strategy and subsequent user behaviors, and investigated which behaviors are more likely to lead to successful recovery. Finally, we also investigated the impact of the non-understanding recovery policy on performance. We started with the assumption that both local recovery and global dialog performance can be improved by using a better recovery policy. To validate this hypothesis, we designed a between-groups experiment in which we investigated performance differences between an uninformed recovery policy (i.e. system picked randomly between recovery strategies) and a "smarter" policy implemented by a human, in a wizard-of-oz setup. The experiment, analysis and corresponding results and observations are presented in detail in section 8.3.

Next, in section 8.4, we turn to the third question. We proposed a novel, online approach for learning recovery policies from data. In an initial experiment conducted with Let's Go! Public, a deployed spoken dialog system that provides bus schedule information, the proposed approach led to statistically significant improvements in recovery performance over the current heuristic policy.

We begin by briefly reviewing other related work and solutions commonly encountered for recovering from non-understandings.

8.2 Related work

8.2.1 Non-understanding recovery strategies

A relatively large number of strategies for recovering from non-understandings have been proposed and are commonly used in different spoken language interfaces. They range from generic strategies such as notifying the user that a non-understanding has occurred, repeating the system prompt, asking the user to repeat or rephrase, to context-specific and problem-specific strategies such as providing different levels of help, telling the user to speak louder or softer, or to move to a quieter place.

Although these strategies are commonly used, our current knowledge about their relative advantages and disadvantages is fragmentary at best. A number of user studies [19, 26, 39, 42, 113, 115, 132, 138, 141] have been conducted to empirically investigate such recovery strategies. In addition, different voice interface experts have also established and published various sets of "best practices" or "guidelines", from an industry perspective [5, 46]. Although progress has been made, the emerging picture is incomplete, and sometimes even inconsistent. For instance, there is wide agreement that repeating (especially multiple times) the exact system prompt is not a good error recovery strategy, as it leads to increased user frustration, which in turn correlates with poor recognition [39, 100]. At the same time, contradictory advice and evidence can also be found. For instance, in [39] it was found that strategies including apologies were associated with lower incidence of frustration in the user responses; the voice interface guidelines described in [5] argue however the opposite. We believe that to a large extent, the difficulties we are facing in assembling a coherent picture stem from the fact that the trade-offs between these strategies are in fact context- (i.e. domain-, task-, user-population-) dependent.

In more recent work, a number of researchers have proposed and investigated in more detail more specialized non-understanding recovery strategies centered on extracting more information from the non-understood utterance. In [42] Gorrell et al. describe and implement a targeted-help strategy based on classifying the non-understood utterance into one of several classes. In [88] Raux et al propose a method to automatically generate confirmation prompts that are close to the non-understood user utterance, and at the same time fall within the system's language model and grammar. In [35], Filisko and Seneff describe a speak-and-spell approach to acquiring city names; this can in general be a very useful non-understanding recovery strategy if the user is trying to specify an out-of-vocabulary word. Additionally, a number of other strategies leveraging other modalities have been proposed and evaluated in the context of multimodal systems [79, 97, 98, 119].

To identify additional recovery strategies, several researchers have turned to wizard-of-oz studies aimed at discovering how humans would handle speech recognition errors. In these studies, instead of hearing the actual user utterance, the wizard receives the text of the recognition result, potentially annotated with confidence information. Zollo shows that under these settings wizards tend to provide large amounts of feedback, even when speech recognition works perfectly [141]. Zollo's study identifies a number of positive and negative feedback strategies: wh-replacement of a missing or erroneous word (e.g. "how many people are where?"), attempts to salvage a correctly recognized word (e.g. "what about the helicopter?"), explicitly verifying of a turn (e.g. "you are ready to begin…"), repeating part or all of the content of the user's utterance, using simple acknowledgements, etc. The majority of the strategies revealed by Zollo's study have no direct equivalent in current spoken dialog systems. In a similar study in a map domain, Skantze shows that wizards tend **not** to signal non-understandings, but rather try to advance the dialog by other means such as asking different task-related questions [115]. Overall, these studies have confirmed that humans employ a larger repertoire of conversational error handling strategies when faced with uncertainties stemming from poor speech recognition.

In our own effort to add to this body of knowledge, we have conducted an empirical investigation of 10 non-understanding recovery strategies; the results are described in detail and compared to previous observations reported in the literature in section 8.3.

8.2.2 Non-understanding recovery policies

8.2.2.1 Heuristic approaches

Most spoken language interfaces use only a limited number of non-understanding recovery strategies in conjunction with simple heuristic policies. The policies are generally designed by a system author, based on their empirical, domain-specific observations or on various guidelines [5, 46] and rules-of-thumb. For instance, it is generally agreed [5, 39, 50, 138] that a good policy must avoid repeating the same recovery strategy over and over. A system might therefore apologize and repeat its question on the first non-understanding, provide more (targeted) help on the second non-understanding, and transfer the user to a human operator if a third consecutive non-understanding occurs. This approach, in which the system tries different strategies on consecutive non-understandings (and perhaps gives up after a number of failed attempts), is quite common. It has been referred to as "progressive assistance" in [138], and as the "three-strikes-and-you're-out" approach in [5].

The problem with heuristic policies is that they are static, and generally under-informed. They use limited amounts of information (such as the number of non-understandings in the current segment) to determine which recovery strategy should be engaged. The system's behavior is based on prior, generic observations rather than on the particular characteristics of the current error or of the domain and environment in which the system operates.

8.2.2.2 Reinforcement learning based approaches

In an effort to provide a more principled and adaptive solution to the problem of developing dialog control policies, a number of researchers have turned their attention to learning based approaches. A technique that has received a lot of attention recently is reinforcement learning [67, 107, 108, 114]. The dialog management problem (or a subset thereof) can be reformulated in terms of a Markov Decision Process (MDP), and reinforcement learning techniques are used to derive an optimal dialog control policy from a training corpus of dialogs. For a detailed overview of this approach, see [114].

The approach has produced successful results in a number of small domains. For instance, Singh et al report on using reinforcement learning to optimize the performance of NJFun, an information access spoken dialogue system [114]. An initial version of the system was used to collect a corpus of exploratory dialogues to serve as training data. The rewards were defined based on a binary measure of task completion, and various choices regarding the type of initiative and confirmation and clarification actions were explored. In the subsequent evaluations of the learned control policy, NJFun showed significant improvements not only in the reward measure for which the optimization was performed, but also on other objective performance metrics. The learned policy outperformed other fixed, handcrafted policies commonly encountered in the literature.

Since the non-understanding recovery policy is a subset of the larger dialog control policy, a reinforcement learning based approach for this problem can be in principle envisioned. Such an approach could provide several advantages: it would allow a system to learn optimal behavior from experience and therefore adapt to the characteristics of particular domains and to slow changes in these characteristics; it could handle delayed feedback, an important feature given the temporal aspects of human-computer interaction; finally, the approach would stand on a solid theoretical foundation. Unfortunately, the reinforcement learning techniques proposed to date suffer from a number of shortcomings that have prevented their use in large scale, practical spoken dialog systems (for an in-depth discussion, see [82].)

Perhaps the most important limitation with respect to developing non-understanding recovery policies is the lack of scalability. As the state- and action-spaces increase, significantly larger amounts of training data are required to estimate model parameters reliably enough and converge on a policy. In part, the difficulties stem from the fact that approach performs a global optimization and takes into account the temporal dimension of the interaction (the number possible paths in the state/action space that need to be explored grows exponentially with the time horizon.) Several approaches have been proposed to alleviate this problem. Typically, the state-space representation is

reduced by using an abstraction of the full information-state of the system [114]. Other researchers have proposed obtaining the required training data through user simulation [67, 108]; later work has however pointed important limitations and pitfalls in simulation-based training [82, 106]. A reduced yet informative state-space is difficult to design in the context of the non-understanding recovery policy problem. As we shall see later, in subsection 8.4.2.3, the set of state features that could carry potentially relevant information for choosing between strategies is very large (e.g. several hundreds); the relative importance of these features is not well known a priori. Furthermore, the cardinality of the action space is dictated by the number of non-understanding recovery strategies under consideration; ideally we would like to build policies that operate over dozens or even more recovery strategies. In contrast, current reinforcement learning based approaches to dialog control are tractable only for a limited number of actions. For instance, in NJFun [114], 2 actions were available in each state; in [108] the total number of actions in the model was 6, but the number of actions available in each state was smaller (2-4). The current lack of scalability renders reinforcement learning based approaches intractable in the context of large, practical spoken dialog systems.

In the work later described in section 8.4, we propose an online, supervised learning approach that, instead of aiming for a global optimum, focuses on improving local recovery performance. In the proposed approach, the system still learns a policy from data in an on-line fashion, and balances between exploration and exploitation. By focusing on a local optimization, we can develop a scalable solution to this problem, in which the system exploits a large set of features (hundreds) to select between a large number of recovery strategies (10 in this case). Initial experiments with a deployed spoken dialog system indicate that the proposed approach leads to significant improvements in the non-understanding recovery rate.

8.3 An empirical investigation of non-understanding recovery strategies and policies

In this section, we investigate ten non-understanding recovery strategies, and two different recovery policies. We begin by describing the experimental setup for the user study that was used to collect the data for this analysis. Next, in subsection 8.3.2 we briefly describe the collected corpus[19]. Then, in subsection 8.3.3 we analyze the performance of each strategy, and in subsection 8.3.4 we analyze the user responses to each strategy. Finally, in subsection 8.3.5 we investigate the effects of different recovery policies on the performance of individual recovery strategies and on global dialog performance. Finally, we summarize the results from this analysis and present a number of concluding remarks in subsection 8.3.6.

8.3.1 A wizard-of-oz data collection experiment

The data collection experiment was designed with two primary goals in mind. First, we wanted to investigate the performance of different non-understanding recovery strategies and analyze subsequent user behaviors, in an effort to develop a better understanding of these strategies. Second, we wanted to validate the hypothesis that the performance of non-understanding recovery strategies can be improved by engaging them at the right time, i.e. by using a good recovery policy. To this end, we designed a user study with two conditions: `control` and `wizard`. Participants in the `control` condition interacted with a system that used an uninformed recovery policy: each time a non-understanding occurred, the system randomly chose one of the ten recovery strategies. Participants in the `wizard` condition interacted with a version of the same system where the policy was implemented at runtime by a human operator (in a wizard-of-oz setting). We begin by describing the sys-

[19] Parts of this corpus have been used for other experiments reported in this dissertation: the error source analysis from Chapter 2, the confidence annotation models from Chapter 5, the belief updating models from Chapter 6 and the error-cost assessment models from Chapter 7.

tem and the set of non-understanding recovery strategies. Then, we discuss in more detail the experimental design. Finally, we describe the participants and the experimental procedure.

8.3.1.1 System

The data collection experiment was conducted using RoomLine, a mixed-initiative, telephone-based spoken dialog system that can assist users in making conference room reservations. The system, described in detail in subsection 3.4.1 from Chapter 3, has access to live information about the schedules and characteristics (e.g. size, location, audio-visual equipment, etc.) of 13 conference rooms in two buildings on campus: Wean Hall and Newell Simon Hall. To make a room reservation, the system finds the list of available rooms that satisfy an initial set of user-specified constraints, and engages in a follow-up negotiation dialog to present this information to the user and identify which room best matches their needs. A sample conversation with the RoomLine system is available in Appendix A.

8.3.1.2 Non-understanding recovery strategies

The system was equipped with ten different strategies for recovering from non-understandings, described and illustrated in Table 38. The strategies represent a subset[20] of the larger repertoire of non-understanding recovery strategies available in the RavenClaw dialog management framework. A number of these strategies, such as asking the user to repeat or rephrase, reissuing the system prompt or providing various levels of help are often encountered in other spoken dialog systems. Two strategies we would like to draw the reader's attention upon are `Yield` and `Move On`. In the `Yield` strategy, the system remains silent, as if it did not hear the user's response, and hence implicitly signals a communication problem. In the `Move On` strategy, the system ignores the current non-understanding altogether and tries to advance the conversation by moving on to a different question. Note that this second strategy is available only at certain points in the dialog, where an alternative dialog plan for achieving the same goals is present. For instance, in the case illustrated in Table 38, the `Move On` strategy gives up on trying to find whether the user wants a small or a large room, and starts suggesting rooms one by one. In another case, the system tries to advance the dialog by using a simpler question, for instance asking "For which day do you need the room?" instead of "How can I help you?"

8.3.1.3 Experimental design

The data collection experiment was designed as a between-groups user study, with two conditions: `control` and `wizard`.

Participants in the `control` condition interacted with a version of the RoomLine system that used an uninformed policy to engage the non-understanding recovery strategies: each time a non-understanding occurred, the system randomly chose one of the ten available strategies.

Participants in the `wizard` condition interacted with a modified wizard-of-oz version of the same system. In this version, each time a non-understanding occurred a human wizard decided which of the ten recovery strategies should be used. In all other aspects, this system was identical with the system used in the `control` condition. The wizard had live access to the user's speech. Several other system state variables were presented to the wizard via a graphical user interfaces (e.g. current recognition result, confidence score, semantic parse). When a non-understanding occurred, the wizard selected which strategy should be used through the GUI, and the decision was forwarded back to the system. The wizard had to make this decision during a relatively short time interval (1-2 seconds) in order to maintain the illusion that users were interacting with an autonomous system. A single wizard

[20] The current, full set of non-understanding recovery strategies in the RavenClaw dialog management framework was discussed earlier in subsection 4.4.1 from Chapter 4. The 10 non-understanding recovery strategies shown in Table 38 were the only ones available at the time this investigation was run.

	S: Would you like a small room or a large one? U: `[Non-Understanding]`
1	**Ask Repeat (AREP)** [system asks the user to repeat the non-understood utterance] S: Can you please repeat that?
2	**Ask Rephrase (ARPH)** [system asks the user to rephrase the non-understood utterance] S: Can you please try to rephrase that?
3	**Repeat Prompt (RP)** [system repeats the previous prompt] S: Would you like a small room or a large room?
4	**Detailed Repeat Prompt (DRP)** [system repeats a more verbose version of the previous prompt] S: I'm sorry, I don't think I understood you correctly. Right now I'm trying to find out if you would prefer a small room or a large one.
5	**Notify Non-understanding (NTFY)** [system notifies the user that a non-understanding has occurred] S: Sorry, I didn't catch that …
6	**Yield Turn (YLD)** [system remains silent, and thus implicitly notifies the user that a non-understanding has occurred] S: Ø
7	**Move On (MOVE)** [system advances the task by moving on to a different question] S: Sorry, I didn't catch that. One choice would be Wean Hall 7220. This room can accommodate 20 people and has a whiteboard and a projector. Would you like a reservation for this room?
8	**You-Can-Say (YCS)** [system tell the user what he or she can say at this point in the dialog] S: Sorry, I didn't catch that. Right now I'm trying to find out if you would prefer a small room or a large one. You can say 'I want a small room' or 'I want a large room'. If the size of the room doesn't matter to you, just say 'I don't care'.
9	**(Terse)-You-Can-Say (TYCS)** [a terser version of the You-Can-Say strategy] S: Sorry, I didn't catch that. You can say 'I want a small room' or 'I want a large room'. If the size of the room doesn't matter to you, just say 'I don't care'.
10	**Full Help (HELP)** [system provides a longer help message which includes an explanation of the current state of the dialog, as well as what the user can say at this point] S: I'm sorry, I don't think I understood you correctly. So far I have found five conference rooms available matching your constraints. Right now I'm trying to find out if you would prefer a small room or a large room. You can say 'I want a small room' or 'I want a large room'. If the size of the room doesn't matter to you, just say 'I don't care'.

Table 38. Ten non-understanding recovery strategies in the RoomLine system

(the author of this dissertation) was used throughout the whole experiment. The wizard had very good knowledge of the system's functionality and of the domain.

The experimental design described above satisfies the two needs outlined at the beginning of subsection 8.3.1. On one hand, we wanted to be able to comparatively evaluate the ten recovery strategies, when engaged in an uninformed fashion. This analysis can be performed based on data collected in the `control` condition. The results are discussed in detail in subsections 8.3.3 and 8.3.4. At the same time, we wanted to investigate whether or not a better recovery policy (implemented in this case by the human wizard) can significantly improve performance. The results of this comparative analysis are presented in subsection 8.3.5.

At this point we would like to briefly comment on the decision to give the wizard full access to the live user speech. This puts the wizard in an apparently privileged position when compared to a system that would have to make the same recovery decisions: the system does not accurately know what the user says, especially during non-understandings. However, recall that our goal is simply to investigate whether a better recovery policy exists and can improve performance, and **not** to establish upper bounds or to prove that this particular policy can be learned or implemented by the system.

Note that the experimental design we have followed does not allow us to establish an absolute upper-bound on the performance of a recovery policy that uses the given set of strategies. This is an interesting and difficult problem, which we have not pursued here. Another interesting baseline,

again not afforded by our experiment, is the human "gold standard" as defined by Paek in [80]. The goal standard recovery policy would be implemented by a human wizard that has access only to the same variables as the dialog manager has at run-time, i.e. recognition results plus all other features available on-line. If computed, the absolute upper-bound and "gold standard" for recovery performance could also help us better understand how much potential for improvement exists.

In this experiment we have simply focused on showing that a better policy exists. The rationale for not computing the "gold standard" (i.e. for giving the wizard access to the live user speech) was as follows. Without access to the user's speech, the decision making task might have been too difficult for the wizard, especially given the response-time constraints. In this case, a negative result, i.e. the lack of detectable differences in the performance of the policies in the `control` and `wizard` conditions, would not be very informative. On the other hand, a negative result obtained when the wizard has full access to the user's speech would cast more serious doubts about the possibility of improving performance through a better recovery policy. Conversely, we believe that the detection of a significant recovery performance gap between the control and wizard conditions would justify pursuing algorithms for developing better policies (note that the absolute upper-bound is higher than the wizard policy performance in our experiment.)

8.3.1.4 Participants

46 subjects, mostly undergraduate students and staff personnel on campus, participated in the data collection experiment. The participants had only marginal prior experience with spoken language interfaces; some of them had previously interacted with phone-based customer-service interactive systems. We randomly assigned the participants into two groups corresponding to the `control` and `wizard` conditions. At the same time, a balance was maintained between groups in terms of the participants' gender and whether or not their first language was north-American English. Each group had 1 female non-native speaker, 11 female native speakers, 5 male non-native speakers and 6 male native speakers.

8.3.1.5 Tasks and experimental procedure

Each participant attempted a maximum of 10 scenario-based interactions with the system, within a set time period of 40 minutes. The same 10 scenarios were presented in the same order to all participants. The scenarios were designed to cover all the important aspects of the system's functionality

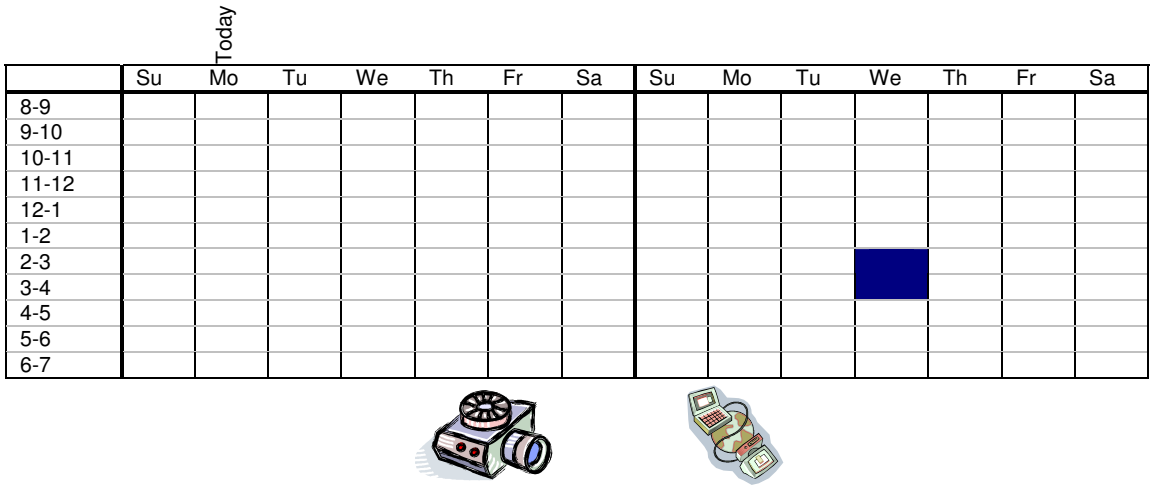


Figure 94. Sample room reservation scenario

and had different degrees of difficulty. To avoid language entrainment, the scenarios were presented graphically. For instance, in the example illustrated in Figure 81 the user was required to make a room reservation for the following Wednesday from 2 to 4 p.m. The room had to have a projector and a network connection. The meaning for the various icons was explained to the participants in a brief introduction prior to their interactions with the system.

After completing their interactions with the system, the participants filled a SASSI evaluation questionnaire [55] containing 35 questions grouped in 6 factors: response accuracy, likeability, cognitive demand, annoyance, habitability, and speed. In addition, participants were asked to describe what they liked most, what they liked least and what would be the first thing they would change in the system.

8.3.2 Data

The corpus of dialogs collected in this experiment (including both the `control` and `wizard` conditions) contains 449 sessions and 8278 user turns. In Table 39 we present a number of additional descriptive statistics. Since pronounced differences exist on a large number of metrics between native and non-native users, we report the breakdown of these figures in the two populations.

Statistic	Total	Native	Non-native
# Subjects	46	34	12
# Sessions	449	338	111
# Turns	8278	5783	2495
Word-error-rate	25.7%	19.7%	39.8%
Concept-error-rate	35.6%	26.2%	57.5%
% Non-understandings	17.0%	13.4%	25.2%
% Misunderstandings	13.5%	9.6%	22.4%
Task success rate	75.1%	85.2%	44.1%

Table 39. Overall corpus statistics

Following the experiment, the user speech data was orthographically transcribed by a human annotator, and subsequently checked by a second annotator. The transcriptions include annotations for various human and non-human noises in the audio signal. Based on these transcriptions, a number of additional annotations were created. At the turn level, we manually labeled:

- **Concept transfer** and **misunderstandings**: each user turn was annotated with the number of concepts that were correctly and incorrectly transferred from the user to the system; each turn with at least one incorrectly transferred concept was automatically labeled as a **misunderstanding**.
- **Transcript grammaticality**: each user turn was manually annotated as either **in-grammar**, **out-of-grammar**, **out-of-application-scope** or **out-of-domain**; these labels were used in an analysis of error sources, described earlier in Chapter 2.
- **User responses to non-understandings***:* the user response types following non-understandings were labeled according to a tagging scheme first introduced by Shin and Narayanan [113]; more details are available in subsection 8.3.4, where we investigate user behaviors following non-understanding recovery strategies.
- **Corrections**: each turn in which the user was attempting to correct a system error was flagged as a correction, as in [121]. These annotations were used in the analysis of user responses to explicit and implicit confirmation strategies, described previously in subsection 6.4.3 from Chapter 6.

In addition, at the session level, we manually labeled **binary task success**. A task was considered successfully completed if all the criteria for the room reservation described by the corresponding scenario were correctly satisfied.

8.3.3 Performance of non-understanding recovery strategies

We now comparatively analyze the performance of the ten non-understanding recovery strategies. This analysis was performed using only the data from the `control` condition. Recall that in this condition, the strategies were engaged by an uninformed policy (i.e. randomly), and therefore they were on equal footing. Later on, in subsection 8.3.5 we discuss and compare the performance of the same strategies, when engaged by the wizard's recovery policy.

We begin by introducing a number of metrics for evaluating recovery performance in subsection 8.3.3.1. Then, in subsection 8.3.3.2 we present the results of the comparative analysis using these different metrics.

8.3.3.1 Metrics for evaluating non-understanding recovery performance

To our knowledge no traditional, well-established metrics exist in the community for evaluating the performance of non-understanding recovery strategies. We therefore propose a number of metrics, which we describe below. The metrics are local in nature; each of them evaluates various characteristics of the user response following the system's attempt to recover from a non-understanding.

The first metric we considered was the **recovery rate (RR)**. This metric takes into account whether or not the next user turn following the system's non-understanding recovery strategy is correctly understood by the system. If the next turn is correctly understood, i.e. it is not a misunderstanding and it is not a non-understanding, then we say that the strategy has successfully recovered. The **recovery rate** is then defined as the ratio of successful recoveries with respect to the total number of attempts to recover. The underlying variable in this metric is binary: the next turn is either correctly understood or not. The metric does not therefore take into account the magnitude or the costs of potential errors in the follow-up user response. Nevertheless, this metric provides a first order estimate of recovery performance. It is easy to understand and interpret, and, because of its low variance, is especially useful when only a small number of samples is available for evaluation.

The recovery rate metric introduced above provides a first, high-level approximation for the performance of the recovery process. However this metric has a number of drawbacks. First, it is rather coarse: it only takes into account whether or not the next user turn is correctly understood, and does not distinguish between different types of errors in the opposite case. For instance, if the user turn following a recovery attempt is a misunderstanding, the system acquires incorrect information. In this case the penalty should be higher than if the next turn was another non-understanding, because, as we have seen in Chapter 2, misunderstandings are in general more costly than non-understandings. Second, the recovery rate is somewhat inappropriate for measuring the performance of the `Move On` strategy. When this strategy is engaged, the system moves on to a different question. It does not really solve the current problem, and an alternative dialog path might have a higher cost than the current one. Last, the recovery rate does not take into account the time elapsed during recovery: some strategies use shorter prompts and therefore might recover (or fail) faster than others. In general, fast recovery is desirable in task-oriented dialog, and ideally we would like to take this into account.

To compensate for these deficiencies and construct a more accurate image of recovery performance, we define three additional, incrementally more refined metrics: **recovery word-error-rate**, **recovery concept utility**, and **recovery efficiency**.

The first refined metric is the **recovery word-error-rate (RWER)**. Instead of looking at whether the next turn is correctly understood or not, we compute and average the word-error-rate for the user turns following non-understanding recovery attempts. This metric captures in more detail the magnitude of the speech recognition errors in the user responses following the system's recovery strategy.

In a spoken dialog system, we are more interested in the correctness of the concepts acquired by the system rather than the correctness of the recognition process per se. The second refined metric we used, **recovery concept utility (RCU)**, operates at the concept level. Instead of considering whether or not the next user turn is correctly understood by the system, we actually

count how many concepts were correctly (CTC) and incorrectly (ITC) acquired by the system in that turn. Additionally, we estimate the average utility of correctly and incorrectly acquired concepts ($Util_{CTC}$ and $Util_{ITC}$) via a data-driven cost-assessment model that relates the average number of correctly and incorrectly acquired concepts to task success. (This type of model has been discussed in detail earlier, in subsection 2.3.1 from Chapter 2 and in section 7.4 from Chapter 7.) The model constructed from the collected corpus revealed that, in the RoomLine domain the average utility of a correctly acquired concept ($Util_{CTC}$) is 7.81, and the average utility (cost) of an incorrectly acquired concept ($Util_{ITC}$) is -7.19. Recovery concept utility is therefore computed as follows:

$$RCU = Util_{CTC} \cdot CTC + Util_{ITC} \cdot ITC$$

$$RCU = 7.81 \cdot CTC - 7.19 \cdot ITC$$

Because it takes the domain-specific costs for correct and incorrect concepts into account, this metric is more appropriate than the traditional concept-error-rate.

Finally, the last metric we considered was **recovery efficiency (RE)**. This metric is similar to recovery concept utility. In addition, it also normalizes for the amount of time spent by the system during recovery. The motivation behind this metric is that some recovery strategies use shorter prompts than others, and therefore might succeed (or fail) faster. To normalize for the amount of time spent during recovery, we compute the number of concepts (correct and incorrect) we would expect the system to acquire on average during that time interval. We then subtract these numbers from the number of correct and incorrect concepts the system did acquire in the follow-up user turn. Recovery efficiency is therefore defined as follows:

$$RE = Util_{CTC} \cdot (CTC - t \cdot r_{CTC}) + Util_{ITC} \cdot (ITC - t \cdot r_{ITC})$$

where t is the time elapsed between the original non-understanding and the next user turn, and r_{CTC} (and r_{ITC}) are the average rates (per second) of acquiring correct (and incorrect) concepts during non-understanding recovery segments. In other words, during the amount of time t the system spent in its attempt to recover, we would expect to obtain on average $t \cdot r_{CTC}$ correct concepts and $t \cdot r_{ITC}$ incorrect concepts. We subtract these from the actual number of correct (CTC) and incorrect (ITC) concepts obtained in the user response, and then take the corresponding utilities into account. The values for r_{CTC} and r_{ITC} are estimated from the collected data. In the RoomLine domain, they are r_{CTC}=5.41e-5 concepts/sec, and r_{ITC}=2.16e-5 concepts/sec. The formula for recovery efficiency becomes:

$$RE = 7.81 \cdot (CTC - t \cdot 0.0000541) - 7.19 \cdot (ITC - t \cdot 0.0000216)$$

Next, we present results from the comparative analysis of the ten non-understanding recovery strategies, using the four performance metrics introduced above.

8.3.3.2 A comparative analysis of non-understanding recovery strategies

We first computed the average non-understanding recovery rate for each of the ten recovery strategies. The resulting performance for each strategy, together with the 95% binomial confidence intervals for these estimates, is illustrated in Figure 95.

To analyze whether or not there are statistically significant differences between the mean recovery rates of the 10 strategies, we performed an overall analysis of variance for binary response variables, i.e. logistic ANOVA. This analysis confirmed that statistically significant differences exist between the means (p=0.000037).

Next, we used logistic ANOVAs to compare each pair of strategies individually. As we have already seen in Table 39, pronounced differences in performance exist between the native and non-native users. As a consequence, we added whether or not the speaker was native as a factor in the pair-wise ANOVAs; this added factor explains a significant amount of the observed variance in the recovery rate, and increases the sensitivity of the statistical analysis. The results are illustrated in

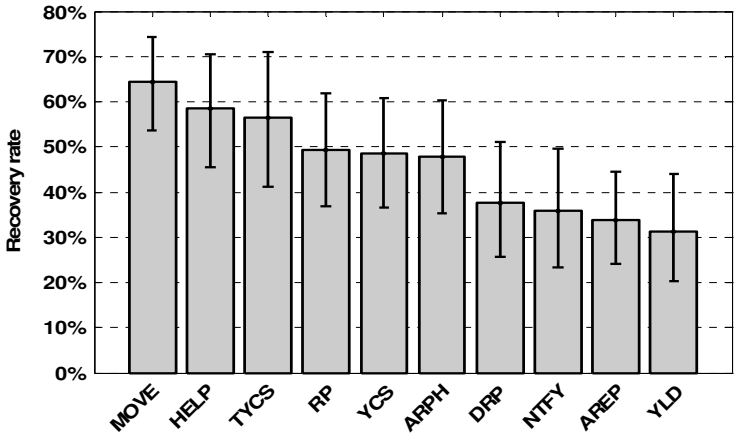


Figure 95. Individual strategy recovery rates (uninformed policy)

Table 40: each cell contains the ratio of the non-understanding recovery rates between the strategies in the corresponding row and column. Since 45 pair-wise comparisons were performed, the p-values need to be corrected for multiple comparisons. The correction was performed using the false-discovery-rate method [6]. This method allows us to compute the expected rate of false-detections among the detected significant differences. The false-discovery-rate (FDR) for each result is illustrated by the gray shading in each cell. For instance, we expect that 5% of the 10 cells with FDR=0.05 do not actually represent significant differences. Similarly, we expect that 10% of the 18 cells with FDR=0.10 and 51% of the 21 cells with FDR=0.15 are actually not statistically significant differences.

Although significant differences cannot be established for every strategy pair, the detected differences allow us to identify a partial ordering. The `Move On`, `Help` and `(Terse)-You-Can-Say` strategies occupy the top 3 positions, with no statistically significant differences detectable between them. In retrospect, this result is not surprising. A number of studies [100, 121] have shown that once an error has occurred, the likelihood of having another error in the next turn is significantly increased (our data also confirms this result). As we go deeper into a spiral of errors, patience runs out, frustration is likely to increase, and the acoustic and language mismatches are likely to become more pronounced. Moreover, the fact that there was a non-understanding in the first place indicates that the system is in a difficult position in terms of decoding the current user intention. When the system abandons the current question and attempts to solve the problem by using a different dialog

			MOVE	HELP	TYCS	RP	YCS	ARPH	DRP	NTFY	AREP	YLD
Move On	MOVE	64.4%	-	1.10	1.14	1.31	1.33	1.35	1.71	1.80	1.91	2.06
Full Help	HELP	58.5%	-	-	1.03	1.19	1.20	1.22	1.55	1.64	1.73	1.87
(Terse)You-Can-Say	TYCS	56.5%	-	-	-	1.15	1.16	1.18	1.50	1.58	1.68	1.81
Repeat Prompt	RP	49.2%	-	-	-	-	1.01	1.03	1.31	1.38	1.46	1.58
You-Can-Say	YCS	48.6%	-	-	-	-	-	1.02	1.29	1.36	1.44	1.55
Ask Rephrase	ARPH	48.6%	-	-	-	-	-	-	1.27	1.34	1.42	1.53
Detailed Repeat Prompt	DRP	37.7%	-	-	-	-	-	-	-	1.06	1.12	1.21
Notify Non-understanding	NTFY	35.7%	-	-	-	-	-	-	-	-	1.06	1.14
Ask Repeat	AREP	33.7%	-	-	-	-	-	-	-	-	-	1.08
Yield Turn	YLD	31.2%	-	-	-	-	-	-	-	-	-	-

Table 40. Comparison of non-understanding recovery rates; the cells show the ratio of the non-understanding recovery rate between the strategy in the corresponding row and column; the shading shows the false-discovery-rate level (FDR=0.15 FDR=0.10 FDR=0.05)

plan, these effects are likely to be attenuated, and chances of correct understanding increase. Similarly, when the system provides help including sample responses for the current question, users might find better ways (from a system's perspective) to express their goals, or they might discover other available options for continuing the dialog from this point.

The high performance of the `Move On` strategy is consistent with prior evidence from a wizard-of-oz study of error handling strategies performed by Skantze [115]. Skantze's study has revealed that, unlike most spoken dialog systems, human wizards often did **not** signal the non-understandings to the user; instead, they asked different task-related questions to advance the dialog. This strategy generally led to a speedier recovery. In the RoomLine system, the `Move On` strategy implements this idea in practice, and the observed performance confirms the prior evidence from Skantze's study. Although not surprising, we find this result interesting, as it points towards a road less traveled in spoken dialog system design: when non-understandings occur, instead of trying to repair the current problem, use an alternative dialog plan to advance the conversation.

The next three strategies, `Repeat Prompt`, `You-Can-Say` and `Ask Rephrase`, form a second tier, all having a statistically better recovery rate than the last 4 strategies. Finally, no significant differences could be detected in terms of recovery rate between the last four strategies: `Detailed Repeat Prompt`, `Notify non-understanding`, `Ask Repeat` and `Yield Turn`.

The discussion above was based on the non-understanding recovery rate metric. In section 8.3.3.1, we introduced three additional, more refined metrics that also take into account the quality of the system's understanding of the follow-up user response, and the time elapsed before this response reaches the system. The ranked list of strategies, according to these additional metrics, is shown in Table 41.B-D. The evaluation on the first two of these metrics, recovery word-error-rate and recovery concept utility (presented in Table 41.B and Table 41.C) leads to results similar to the evaluation based on recovery rate: the `Move On`, `Full Help` and `(Terse)You-Can-Say` strategies are the top performing ones, and `Detailed Repeat Prompt`, `Notify non-understanding`, `Ask Repeat` and `Yield Turn` occupy the bottom positions. The distance between these lists can be measured using

A. Ranking by recovery rate

Ranking by Recovery Rate	Recovery Rate
Move On	64.4%
Full Help	58.5%
(Terse)You-Can-Say	56.5%
Repeat Prompt	49.2%
You-Can-Say	48.6%
Ask Rephrase	48.6%
Detailed Repeat Prompt	37.7%
Notify Non-understanding	35.7%
Ask Repeat	33.7%
Yield Turn	31.2%

B. Ranking by recovery word-error-rate

Ranking by Recovery Word-Error-Rate	Recovery WER
Move On	33.0%
(Terse)You-Can-Say	36.6%
Full Help	39.9%
Notify Non-understanding	41.3%
Repeat Prompt	44.0%
Ask Rephrase	47.1%
Ask Repeat	50.1%
You-Can-Say	51.0%
Yield Turn	52.2%
Detailed Repeat Prompt	53.4%

C. Ranking by recovery concept utility

Ranking by Recovery Concept Utility	Recovery Concept Util.
Move On	3.98
Full Help	3.92
(Terse)You-Can-Say	3.75
You-Can-Say	2.75
Ask Rephrase	2.63
Repeat Prompt	2.19
Yield Turn	1.95
Detailed Repeat Prompt	1.71
Notify Non-understanding	1.51
Ask Repeat	1.47

D. Ranking by recovery efficiency

Ranking by Recovery Efficiency	Recovery Efficiency
Move On	1.43
(Terse)You-Can-Say	1.11
Ask Rephrase	0.75
Repeat Prompt	0.29
Ask Repeat	-0.1
Notify Non-understanding	-0.38
Full Help	-0.4
You-Can-Say	-0.47
Detailed Repeat Prompt	-0.94
Yield Turn	-1.93

Table 41. Ranked performance of ten non-understanding recovery strategies using four evaluation metrics

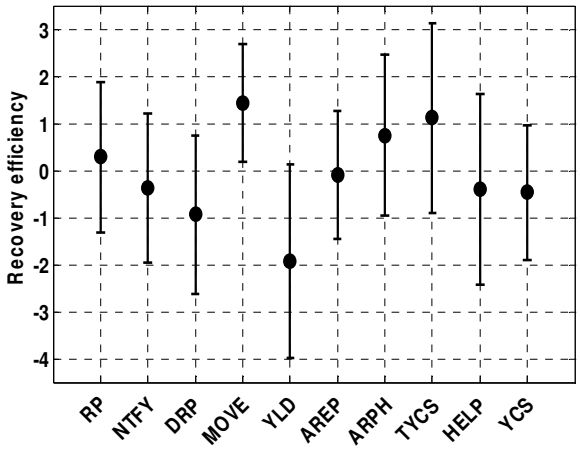


Figure 96. Average non-understanding recovery efficiency for each recovery strategy (under an uninformed policy)

Spearman's rank correlation coefficient [135]. The correlation between the recovery rate ranking and recovery word-error-rate ranking is ϱ=0.7454; the correlation between the recovery rate ranking and the recovery concept utility ranking is even higher ϱ=0.8848.

Once we also take the elapsed time into account (see Table 41.D), the ranking changes more significantly: ϱ is significantly lower, 0.5878. `Move On` remains the top performing strategy. However, due to longer prompts, the more verbose help strategies (`Full Help` and `You-Can-Say`) drop significantly in rank, and other shorter strategies gain relatively (`Ask Rephrase` and `Ask Repeat`.) The recovery efficiency estimates and 95% confidence intervals are illustrated in Figure 96. We repeated the pair-wise comparisons between the strategies using this new metric. This time, Mann-Whitney U-tests were used to compute the p-values, because recovery efficiency is a continuous-score and is not normally distributed; again the p-values were corrected using the false-discovery-rate method. Unfortunately, the larger variance in the recovery efficiency metric (introduced by the time normalization), coupled with the relatively small number of samples available for each strategy, does not allow us to identify many statistically significant differences. At an FDR of 0.10 we find that the `Yield Turn` strategy performs significantly worse than the top 6 strategies, and `Detailed Repeat Prompt` performs significantly worse than the top 3 strategies.

8.3.4 Analysis of user responses to non-understanding recovery strategies

Next, we analyzed the user responses that follow each non-understanding recovery strategy, in an effort to identify which user response-types lead more often to successful recoveries. The analysis was conducted using data collected in the `control` condition, where the strategies were engaged in an uninformed manner.

8.3.4.1 Tagging scheme

To perform this analysis, each user turn that followed a non-understanding was labeled according to a tagging scheme for error segments first introduced by Shin [113]. Subsequently, versions of this tagging scheme have been used by others: for instance, Choularton and Dale [26] used an abbreviated version of Shin's original scheme to analyze a corpus of dialogs from a deployed dialog system for ordering pizza; Raux et al. [89] used it to analyze data collected with Let's Go! Public, a deployed telephone-based system that provides bus route and schedule information.

In this analysis, we used the same tagging scheme as Choularton and Dale [26]. User responses during error segments are classified into one of five categories:

- `repeat`: the user repeats the previous utterance identically (at the lexical level);

- `rephrase`: the user rephrases the same semantic content, but using different lexical choices;
- `change`: the user changes the semantic concepts with respect to the previous utterance;
- `contradict`: the user contradicts the system (often as a barge-in);
- `other`: subsumes response types that do not fall in any of the previous categories (e.g. hang-ups, timeouts, requests for help, etc.)

8.3.4.2 Experimental results

Figure 97 shows the overall distribution of user response types in our dataset. As a reference, we also show the user response type distributions found by Shin in an analysis of the Communicator corpus [113], and Choularton and Dale in an analysis of a deployed system for ordering pizza [26].

Note however that a direct comparison between these experiments is not valid since we only analyzed user responses that followed a non-understanding (any user turn throughout any error segment). The distribution of user response types we observed is nonetheless similar to the previous studies. When faced with non-understandings, users tend to `rephrase` (~45%) more than `repeat` (~20%). One notable difference across the distributions can be observed between the `change` and `contradict` user response-types. We believe this difference is largely due to the fact that we only analyzed the user turns following non-understandings: `contradicts` occur mostly when a system misunderstands.

The larger number of `change`-type responses is to a large extent introduced by the `Move On` strategy (see also Figure 98 and Figure 99). While in Shin's study of the Communicator data a lot of

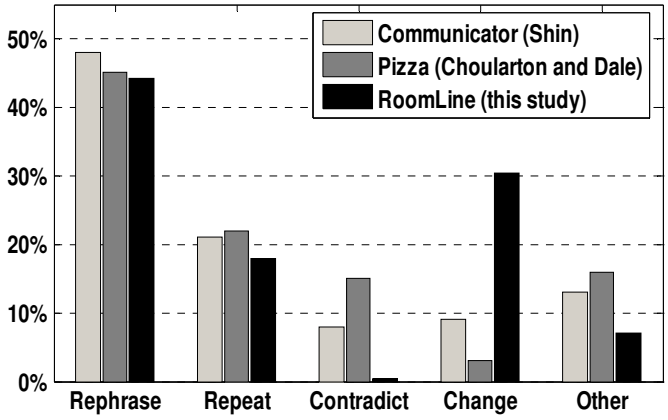


Figure 97. Distribution of user-response types

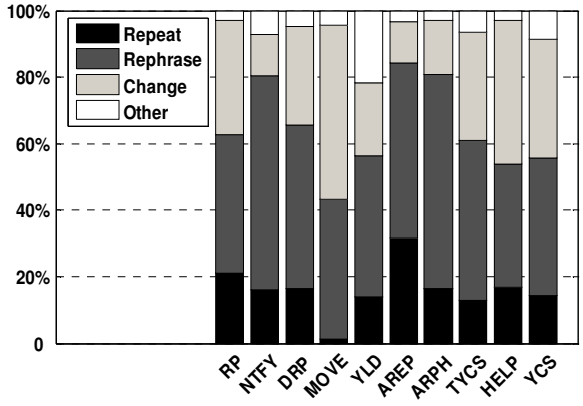


Figure 98. Distribution of user response types by non-understanding recovery strategy

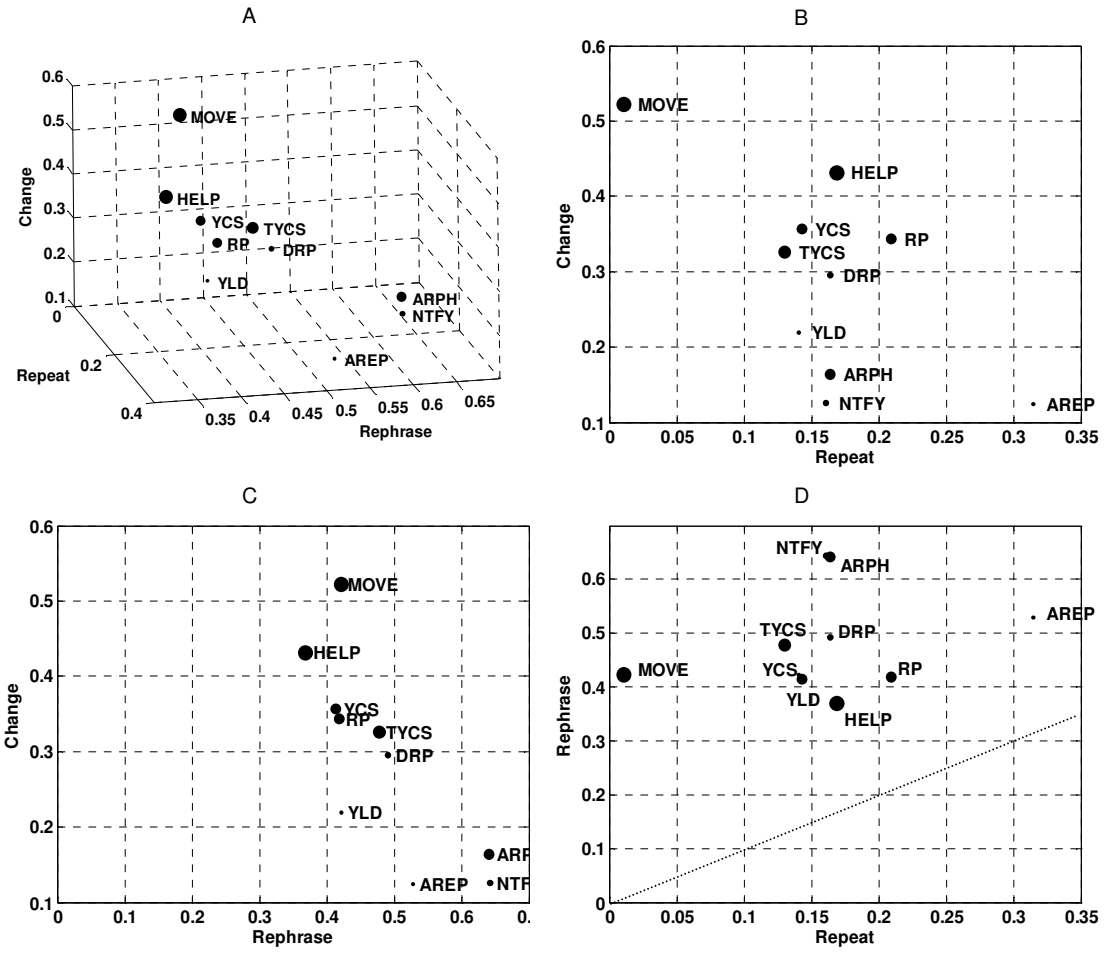


Figure 99. A: 3-dimensional representation of the 10 non-understanding recovery strategies in the space of user response types that they induce. The points' magnitudes reflect the recovery success rates of each strategy. B-D: Projections of the 3-d representation on the 3 planes.

`change` responses occurred because users randomly changed their travel plans to simply advance the dialog, this is not the case in our study. Recall that participants in the study were compensated according to the number of scenarios they managed to complete successfully. The `change` responses in the collected data generally represent valid contributions to the dialog, within the confines of the given scenarios. For instance, consider a scenario where the user needs to reserve a conference room for two hours in the morning. When asked for the start time, the user says *"from 8 a.m. to 10 a.m."* but this leads to a non-understanding. The system asks the user to repeat, and the user responds *"from 9 a.m. to 11 a.m."*. This is a `change`-type response; at the same time, the ultimate user goal (making a 2-hour reservation in the morning) has not changed.

Next, we analyzed the relationship between each strategy and the follow-up user response-types. The distribution of response-types for each strategy is illustrated in Figure 98. Figure 99 also shows a three dimensional representation of the strategies in the space of user response-types. The results indicate that, as expected, `Ask Repeat` leads to the largest number of `repeat`-type responses (31%); the `Move On` strategy leads to the largest number of `change`-type responses (52%); finally, the `Ask Rephrase` and `Notify Non-understanding` strategies lead to the largest number of `rephrase`-type responses (64%). Although the recovery strategies have a significant impact on the distribution of user response-types, this effect is not particularly strong. If we make the assumption that certain types of user responses are more desirable in certain circumstances, the results presented

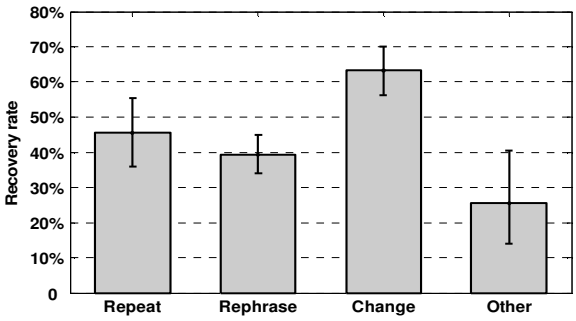


Figure 100. Average recovery rate for different user response types in the RoomLine corpus

above raise the following question: could we control the user response-types even more, for instance by using a more "aggressive" prompting style? (e.g. "Could you repeat what you just said?" instead of "Can you please repeat that?")

To a certain degree, the patterns we have observed in the RoomLine corpus are consistent with prior observations by Goldberg et al. [39] in a study of NIST 2000 Communicator evaluation corpus. For instance, Goldberg found that users were most likely to rephrase (84%) after a `Notify Non-understanding` prompt. In addition Goldberg also found that users are more likely to rephrase rather than repeat their utterances, regardless of the non-understanding recovery strategy used. This again is corroborated by our observations: in Figure 99.D, all strategies are placed above the line of equal repeats and rephrases.

Last, we analyzed which user response-types are more likely to lead to successful recovery. Figure 100 shows the average recovery rate for each user response-type. The best recovery performance is attained on `change` responses (63%). Together with the large number of `change` responses on the `Move On` and help strategies, this result corroborates the high performance of these strategies, and the discussion from section 8.3.3. Somewhat surprisingly, we were not able to establish a statistically significant difference between the recovery rates of user `repeat` and `rephrase` responses. In this respect, our results conflict with a prior study by Goldberg et al. [39], in which it was found that user rephrases are better recognized and more likely to lead to successful recovery. Moreover, the same analysis performed on the sessions collected in the `wizard` condition shows that in that case `repeat` responses were actually significantly better recognized than `rephrase` responses. Briefly, we believe this last result is explained by the fact that the wizard made intensive use of the `Ask Repeat` strategy, when this strategy was appropriate. This in turn boosted the overall number, as well as the recovery performance, of `repeat`-type responses. This result confirms the intuition that the performance of various non-understanding recovery strategies is sensitive to the context (i.e. domain, task, user-population, recovery policy) in which the strategies are used.

8.3.5 Effect of policy on performance: wizard versus uninformed

So far, we have discussed the performance of individual recovery strategies and the follow-up user responses, when the strategies are engaged in an uninformed manner. In this section, we compare the recovery policy implemented by the human wizard against the uninformed policy. The comparison is performed based on the data collected in the two conditions: `wizard` and `control`. We study the effect of the policy on local recovery performance, as well as on global dialog performance metrics. In addition, we investigate the impact of the non-understanding recovery policy on the performance of individual non-understanding recovery strategies.

8.3.5.1 Effect of policy on local recovery performance and global dialog performance

We measured the impact of the recovery policy on local recovery performance using the four metrics that we have previously introduced in section 8.3.3.1 – **recovery rate (RR)**, **recovery word-error-rate (RWER)**, **recovery concept utility (RCU)** and **recovery efficiency (RE)**.

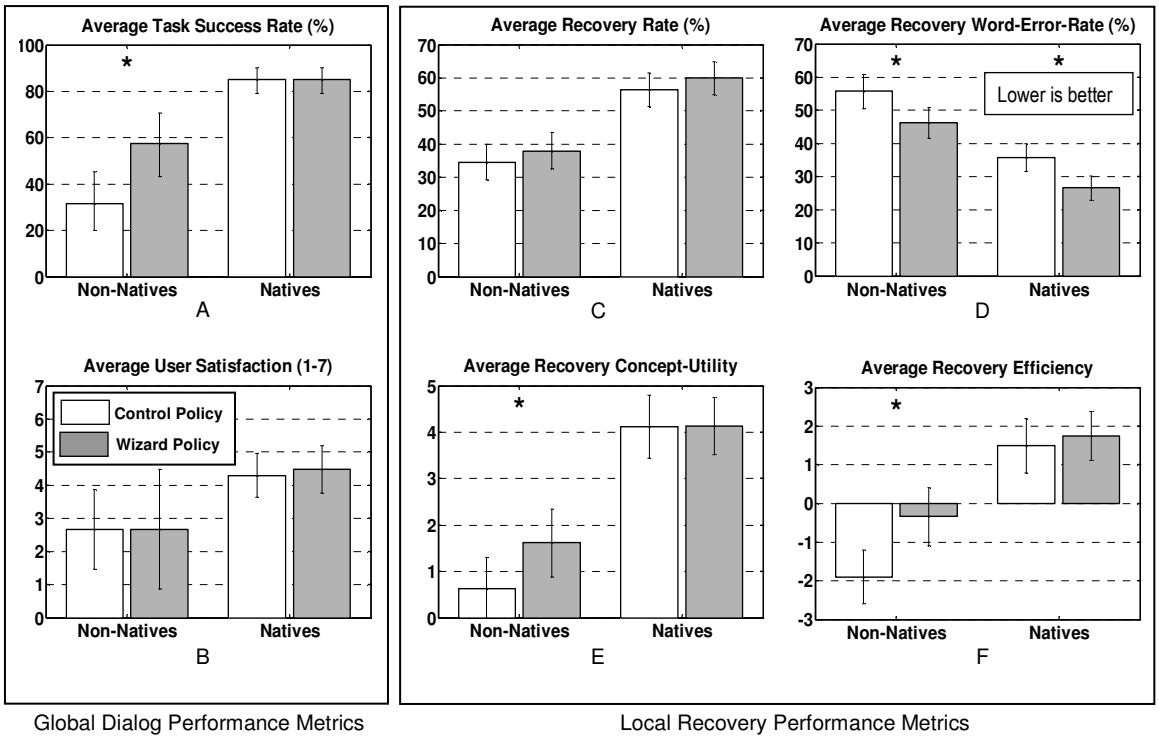


Figure 101. Performance comparison between the wizard and the uninformed recovery policy (* marks a statistically significant difference at $p < 0.05$)

Metric		Overall	Wizard vs Uninformed	Wizard vs Uninformed (only natives)	Wizard vs Uninformed (only non-natives)
Task Success (%)	(a)	75.1	78.5 ≈ 71.7	85.2 ≈ 85.2	57.4 > 31.6
User Satisfaction (1-7)	(b)	3.93	3.87 ≈ 4.00	4.29 ≈ 4.47	2.67 ≈ 2.67
Recovery rate (%)	(c)	48.7	50.1 ≈ 46.5	61.0 ≈ 56.4	37.9 ≈ 34.4
Recovery word-error-rate (%)	(d)	38.9	35.4 < 44.5	26.6 < 35.7	46.4 < 55.7
Recovery concept utility	(e)	2.80	3.01 ≈ 2.58	4.13 ≈ 4.12	1.62 > 0.63
Recovery efficiency	(f)	0.41	0.81 > 0.00	1.74 ≈ 1.50	-0.34 > -1.90

Table 42. Performance comparison between the wizard and the uninformed recovery policy (* marks statistically significant differences at $p < 0.05$)

In addition, to evaluate the impact on global dialog performance, we measured overall **task success** and **user satisfaction**. Task success was defined as a binary variable and manually annotated after the data was transcribed. A task was considered successfully completed if the user made a room reservation that satisfied all the constraints specified in the corresponding scenario. User satisfaction was captured on a 1-7 Likert scale, and was elicited through the SASSI [55] post-experiment questionnaire. The user satisfaction score corresponds therefore to the overall experience the user had with the system.

The results of the comparison are shown in Table 42 and illustrated in Figure 101.A-F. Since performance varies considerably between the native and non-native users, we present the breakdown of the differences in these two populations. In Table 42, the second column shows the overall performance (both groups together); the third column shows the overall differences between the `wizard` and the `control` conditions, while columns 4 and 5 show the differences between these two conditions within the native and non-native populations. The shaded cells mark differences that are statistically significant at a p-value smaller than 0.05. To test for statistical significance we used t-tests

when comparing proportions (e.g. binary task success or recovery rate), and non-parametric Mann-Whitney U-tests for the other continuous-valued performance metrics.

As Figure 101 and Table 42 illustrate, an overall pattern emerges. The wizard policy does indeed lead to statistically significant performance improvements on a number of metrics, but the improvements appear mostly within the non-native population, i.e. in the group of users that had more difficulties using the system. For instance, while no task success improvement can be detected for native users, there is a large task success improvement for non-native users – see Figure 101.A. The average task success rate grows significantly from 31.6% in the `control` condition to 57.4% in the `wizard` condition. This increase bridges half of the original performance gap between native and non-native users in the `control` condition. Despite this increase in task success rate, no statistically significant differences can be detected in user satisfaction – see Figure 101.B; the small number of available samples (one per user) and the large variance of this metric (perhaps also due to different user expectations) preclude a reliable comparison. Nevertheless, the same trend of larger, statistically significant improvements for the non-native users is again observed on the local recovery performance metrics – see Figure 101.C-F. Statistically significant improvements can be detected in the non-native population for three of these metrics: recovery word-error-rate, recovery concept utility, and recovery efficiency.

We believe the explanation for the observed result lies in the fact that it is easier to improve performance when performance is low (in our case, for the non-native users). This result is consistent with our previous analysis of the impact of non-understanding errors on task success described in section 2.3 from Chapter 2.

8.3.5.2 Effect of policy on individual strategy performance

In the previous subsection, we have seen that the wizard recovery policy leads to overall improvements in performance, especially for the non-native users. In this section, we analyze the effect of the policy on the performance of individual recovery strategies. Our initial hypothesis was that, if the strategies are engaged "at the right time", their performance would improve.

Figure 102 shows the number of times each non-understanding recovery strategy was engaged by the wizard. Figure 103 shows the recovery rate for each of the ten strategies, under the two different policies (wizard and uninformed). We found a statistically significant difference ($p=0.0023$, or $p=0.023$ Bonferroni corrected for multiple comparisons [105]) only for the `Ask Repeat` strategy. Note however that `Ask Repeat` is the strategy most often engaged by the wizard. While this strategy ranked 9th when engaged in an uninformed fashion, its performance improved considerably from 33.7% to 53.0% under the wizard policy and is on par with the other top-performing strategies such as giving help, such as `(Terse)You-Can-Say` and `Full Help`, or advancing the task by asking a dif-

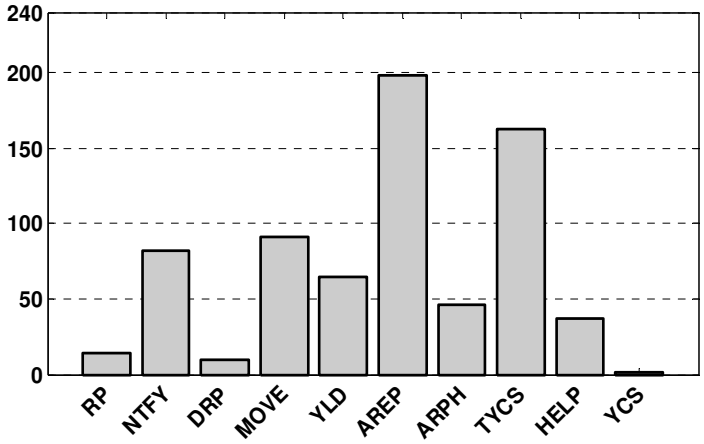


Figure 102. Number of times each strategy was engaged by the wizard

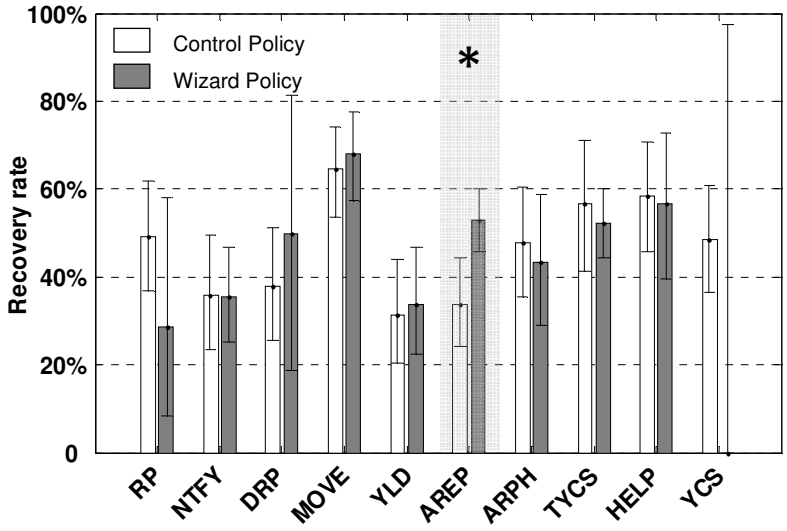


Figure 103. Impact of non-understanding recovery policy on individual strategy performance

ferent question, such as `Move On`. The same improvement in the `Ask Repeat` strategy was also detected on the other three recovery performance metrics.

This result confirms that indeed strategy performance can be improved by the use of a better recovery policy. At the same time, the lack of detectable differences in the other strategies is somewhat unexpected. In retrospect, this result might be explained by the fact that the decision task the wizard had to perform was quite difficult, even with access to the full audio signal. To maintain the illusion that users were interacting with an autonomous system, the wizard had to choose one of ten recovery strategies in a very short time interval: 1 to 2 seconds. This selection task is easier for some of the strategies than for others. Furthermore, a number of strategies, such as `You-Can-Say`, `Repeat Prompt`, and `Detailed Repeat Prompt`, were very rarely engaged by the wizard and as a result the confidence bars on their performance estimates are very wide, and preclude an accurate comparison.

8.3.6 Concluding remarks

In this section, we have described an empirical investigation of ten non-understanding recovery strategies engaged by two different recovery policies. The questions under investigation were: **(1) how do various non-understanding recovery strategies compare to each other?** and **(2) can a good non-understanding recovery policy lead to significant performance improvements?**

We began by introducing four metrics for assessing recovery performance, and comparing the ten strategies across these dimensions. The results show that, when engaged in an uninformed fashion, the best performing strategies in our domain are: (1) advancing the conversation by ignoring the non-understanding and trying an alternative dialog plan (`Move On`), and (2) providing help messages containing sample responses for the current system question. At the same time, generic strategies such asking the user to repeat or notifying that a non-understanding has occurred, did not perform well. An analysis of user responses to various strategies indicates that, while there is an effect of strategy on the subsequent user behavior (e.g. users repeat most when they are asked to repeat, etc.), this effect is not particularly strong. In the future, it would be interesting to investigate to which extent user responses can be controlled, for instance by using a more aggressive prompting style, such as "Could you repeat what you just said?" instead of "Can you please repeat that?"

To a certain degree, our results are in line with previous observations made by others. For instance, the high performance of the `Move On` strategy corroborates prior evidence from a wizard-of-oz study [115] that showed that human operators often do not signal non-understandings, but rather try to advance the task by asking different questions. In the future, we plan to explore in more detail the potential uses of this strategy, as well as its pitfalls. Potential issues to be investigated in-

clude identifying more situations in which this strategy is applicable, studying the extent to which this strategy can be decoupled from the system's task, and developing more appropriate metrics for assessing its performance. Our analysis also corroborated previous observations made by Shin et al. [113], and Choularton and Dale [26] with respect to the overall distribution of user respose-types during non-understanding segments. Furthermore, our data also corroborates findings by Goldberg [39] that users are more likely to rephrase than repeat, after any prompt in an error sub-dialog.

At the same time, it is important to notice that a coherent, consistent image about the performance and effects of various non-understanding recovery strategies is hard to establish. While informative, results regarding the performance of these strategies and the distribution of user responses do not necessarily generalize across domains. For instance, we found conflicting evidence with Goldberg's study [39] on the relative likelihood of recovery for user repeats and user rephrases. The success of various strategies can be strongly influenced by a number of factors such as the nature of the task, the user population, as well as the policy used to engage the strategies. We believe that the solution for successful recovery lies in endowing spoken dialog systems with the capacity to adapt their error handling behaviors to the specific characteristics of the domains and environments in which they operate.

As a first step in this direction, we have investigated whether or not a more informed recovery policy (in our case implemented by a human in a wizard-of-oz setup) can lead to significant performance improvements. The empirical results confirm this hypothesis. The more informed policy led to significant improvements in task success, as well as on a number of other local performance metrics. The improvements occurred mostly within the non-native population, i.e. the group of users that had more difficulties interacting with the system.

In the next section, we turn our attention to the third issue we have outlined in the introduction of this chapter: **how can we develop good non-understanding recovery policies that operate over large sets of recovery strategies?**

8.4 An online, supervised approach for learning non-understanding recovery policies

We have seen in the previous section that a good non-understanding recovery policy can lead to statistically significant improvements in both local and global performance metrics. However, developing good policies is a challenging task. The number of potential strategies the system could engage is relatively large. The trade-offs between these strategies are not always well understood and they probably differ across domains. Ideally, we would like systems to learn from their own experiences, and adapt to the particular characteristics of the domain and environment in which they operate.

In this section, we propose an online, data-driven approach for developing non-understanding recovery policies over a large set of recovery strategies. We begin by describing the proposed method in the next section, 8.4.1. Then, in section 8.4.2 we describe an experiment conducted to evaluate the proposed learning methodology in the context of a real-world, deployed spoken dialog system. Finally, in section 8.4.3 we summarize the results and discuss a number of avenues for further developing and extending this work.

8.4.1 Method

The starting point for the proposed method lies in the observation that certain non-understanding recovery strategies are more likely to succeed in certain circumstances. If we were able to compute the likelihood of success for each recovery strategy at runtime, a policy would be easy to construct: for instance we could choose the strategy with the highest likelihood of success. We therefore propose a method that works in two steps: first, we use a supervised learning approach to construct predictors for the likelihood of success of each individual recovery strategy. Then, we use these predictors at run-time to select which strategy to engage.

Note that we are interested in developing an online solution to this problem: the likelihood-

of-success predictors should be refined as increasingly larger amounts of data become available to the system; the policy should adjust accordingly. As we have argued in the introductory chapter, spoken dialog systems are interactive by nature and operate in shifting environments. Simple changes in the recovery policy might trigger subsequent changes in user behavior. Other independent factors such as learning effects enabled by long-term use of the system might also affect user behavior. In general, we should expect that the underlying distribution of the training data changes through time; an online solution (as opposed to a batch mode approach) could track these changes in the underlying training data and adjust the policy accordingly.

8.4.1.1 Predictors for strategy success

We use logistic regression models to predict the likelihood of success for each recovery strategy. One separate model is constructed for each strategy. Each model predicts whether or not the strategy has successfully recovered, in the sense defined in subsection 8.3.3.1, i.e. the next user turn is correctly understood by the system. For training and evaluation purposes, this information is manually annotated. In fact, the system already knows automatically when non-understandings occur. As a consequence, a semi-automatic approach can be used to create the recovery labels: all the non-understandings followed by another non-understanding are automatically labeled as not-recovered; the remaining non-understandings are inspected and labeled by a human annotator. Later, in subsection 8.4.3, we also outline a possible implicitly-supervised approach for training these predictors.

The features, or the dependent variables in the regression model, capture various aspects of the last non-understanding, as well as information about the current dialog state and about the history of the dialog so far. The full set of features used in these experiments is described in detail later, in subsection 8.4.2.3.

Logistic regression models [76] present a number of advantages over other machine learning techniques in this task. As we have already seen in Chapter 5, in contrast with other discriminative approaches, logistic regression generally produces well-calibrated class posterior probability scores [28, 139]. In other words, the model predictions accurately reflect the probability of success: a strategy will be successful in x% of the cases when the model predicts that the likelihood of success is x. Like for building confidence annotators, this is an important property because we plan to use the model outputs as probability estimates. Second, logistic regression models are sample efficient. This is another desirable property because we plan to learn one separate model for each strategy and only a relatively small number of data-points will be available for training each predictor. A third advantage of logistic regression models is that they can be constructed in a stepwise manner. This allows us to consider a very large number of features; the relevant features are automatically selected and included in the model. Last, logistic regression models can automatically provide the confidence intervals for their predictions, which is an essential prerequisite for the strategy selection method described in the next subsection.

8.4.1.2 Highest-upper-bound strategy selection method

Once we can predict the likelihood of success for each individual recovery strategy, we are left with choosing the method for selecting between the strategies. Ideally, we should choose the strategy with the highest likelihood of success. However, we are interested in developing an approach in which the system learns a recovery policy on-line, through experimentation. As a result, we are faced with an exploration-exploitation tradeoff. The system needs to strike the right balance between using strategies it knows to be successful (exploitation) and gathering more training data for the strategies about which it is still unsure (exploration).

One method for addressing this exploration-exploitation trade-off is to always select the strategy that has the highest upper bound for the estimated probability of success. This **highest-upper-bound selection** method, also known as the interval-estimation, was initially proposed by Kaelbling in [60], and was shown empirically to perform very well in a variery of exploration-exploitation tasks.

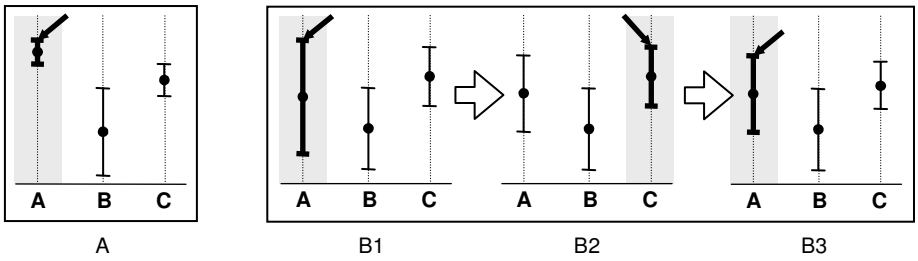


Figure 104. Highest-upper-bound selection between 3 fictitious strategies (A, B, and C)

The intuition behind this method is illustrated in Figure 104. By selecting the strategy with the highest upper bound, the system either chooses a strategy with a high likelihood of success (Figure 104.A), or a strategy with a low likelihood of success but with wide confidence bounds (Figure 104.B1). In the first case, the system is exploiting: it is engaging a strategy that is already known to be successful in that situation. In the second case, the system is exploring. Consider for instance the situation in Figure 104.B1. After strategy A is engaged, a new data-point for training the predictor for that strategy is obtained. As more data becomes available for a given strategy, generally the confidence interval for the corresponding prediction model will shrink. As a result, another strategy (C in this case) will have the highest upper bound, and the system will switch to exploring that strategy (Figure 104.B1 → Figure 104.B2). As more data becomes available for the model predicting the likelihood of success for C, its corresponding confidence interval will shrink more, and again another strategy (A in this case) will be selected – see Figure 104.B2 → Figure 104.B3. In effect, as more data is collected, the confidence bounds on the various predictors iteratively shrink, and the system explores the space of strategies, while at the same time generally engaging strategies that are known to be well-performing.

8.4.2 Experimental results in the Let's Go! Public system

To evaluate the proposed learning methodology, we conducted an experiment with the Let's Go! Public spoken dialog system. In the next two subsections, we describe the system and the set of non-understanding recovery strategies used in this experiment. Then, in subsection 8.4.2.3 we describe the set of features used to construct the individual predictors for the likelihood of success for each recovery strategy. In subsection □ we describe the experimental setup, and finally, in subsection 8.4.2.5 we present the results we obtained.

8.4.2.1 System

The experiment described below was performed in the context of Let's Go! Public [89], a telephone-based spoken dialog system that provides access to bus route and schedule information for the Pittsburgh Port Authority busses. The system, described in more detail in subsection 3.4.2 from Chapter 3, has been available to the general public via the Port Authority customer service line during non-business hours (i.e. 7pm-7am on weekdays and 6pm-8am on weekends and holidays) since March 2005. Throughout this time, the system has serviced over 30,000 calls. On average, the system receives about 50 calls per night.

Given the high traffic volume, and the real-world nature of the application, this system provides an excellent platform for evaluating the proposed online learning method. Because the system services calls only at night, we could label the data and retrain the strategy success prediction models on a daily basis (more details about the learning setup are described later, in subsection □ .)

8.4.2.2 Strategies

During the first year of operation, the system used 5 non-understanding recovery strategies in conjunction with a simple heuristic policy that took into account the number of consecutive non-understandings (for more details, see [89]). Prior to starting the policy learning experiment, we ex-

	S: Where do you want to leave from? U: `[Non-Understanding]`
1	**Ask Repeat (AREP)** [system asks the user to repeat the non-understood utterance] S: Can you repeat what you just said?
2	**Ask Rephrase (ARPH)** [system asks the user to rephrase the non-understood utterance] S: Can you please try to rephrase that?
3	**Repeat Prompt (RP)** [system repeats the previous prompt] S: Where do you want to leave from?
4	**Move On (MOVE)** [system advances the task by moving on with a different question] S: Let's proceed step by step… In what neighborhood is your departure stop?
5	**(Terse)-You-Can-Say (TYCS)** [a terse version of the You-Can-Say strategy] S: For example, you can say, FORBES AND MURRAY, DOWNTOWN, or MCKEESPORT.
6	**(Terse)-You-Can-Say and Restart (TYCS_R)** [a terse version of the You-Can-Say strategy that also told users they can say 'start over' if they want to restart the dialog] S: For example, you can say, FORBES AND MURRAY, DOWNTOWN, or MCKEESPORT, or say 'start over' to restart.
7	**Ask Short Answer (ASA)** [ask the user for a short answer] S: Please use shorter answers because I have trouble understanding long sentences. Where do you want to leave from?
8	**Speak Less Loud (SLL)** [asks the user to speak less loud] S: I understand people best when they speak softer. Where do you want to leave from?
9	**Interaction Tips (IT)** [give general interaction tips to the user] S: Okay, I know this conversation isn't going well. There are things you can try to help me understand you better. Speak clearly and naturally; don't speak too quickly or too slowly. Give short, concise answers. Calling from a quiet place helps. If you'd like to start from scratch, you can say 'start-over' at any time. Where do you want to leave from?
10	**Ask Start Over (ASO)** [asks the user if he/she would like to start over] S: I'm sorry, I'm still having trouble understanding you and I might do better if we restarted. Would you like to start over?
11	**Give Up (GUP)** [give up the dialog and hang up] S: I'm very sorry, but it doesn't seem like I'm able to help you. If your question can wait until 7, you can call back then to speak to a person. I hope that I can be more helpful the next time you call. Thank you for calling the CMU Let's Go bus information system. Goodbye.

Table 43. Non-understanding recovery strategies in the Let's Go! Public system

panded and redesigned the set of non-understanding recovery strategies. The final set is shown in Table 43.

In addition, we designed a set of rules to restrict the circumstances under which each non-understanding recovery strategy can be used:

- after three consecutive non-understandings in the states where the `Move On` strategy is possible, force the `Move On` strategy (i.e. all other strategies become unavailable)
- don't use `Ask Repeat` more than twice in a row (avoids triggering frustration by repetitive behavior;
- don't use `Ask Rephrase` more than twice in a row (avoids triggering frustration by repetitive behavior);
- don't use `Ask Rephrase` if the recognized hypothesis has only one word (it's likely that short utterances cannot be really rephrased);
- don't use `Repeat Prompt` more than twice in a row (avoids triggering frustration by repetitive behavior);

- don't use `(Terse)-You-Can-Say` and `(Terse)-You-Can-Say and Restart` if the last strategy was also `(Terse)-You-Can-Say` or `(Terse)-You-Can-Say and Restart` (avoids triggering frustration by repetitive behavior);
- don't use `(Terse)-You-Can-Say` and `(Terse)-You-Can-Say and Restart` if they were already used three times in this particular dialog state;
- don't use `(Terse)-You-Can-Say and Restart` unless there were already 10 turns in the dialog (ensures the system doesn't tell the user about restarting at the very beginning of the dialog);
- don't use `(Terse)-You-Can-Say and Restart` on the initial "How may I help you?" state (ensures the system doesn't tell the user about restarting at the very beginning of the dialog);
- don't use `Speak Less Loud` if it was already used in the last four turns or if the signal energy level is below a certain threshold;
- don't use `Ask Short Answer` unless the number of words in the recognized hypothesis is above 4;
- don't use `Ask Short Answer` if it was already used in the last 4 turns;
- don't use `Interaction Tips` unless there are at least 4 consecutive non-understandings (this strategy should be used sparingly, as a last resort);
- don't use `Interaction Tips` more than once per dialog;
- don't use `Ask Start Over` unless there are 4 or more consecutive non-understandings, at least 10 turns in the dialog, and the ratio of non-understandings so far is above 0.5 (this strategy should be used sparingly, as a last resort);
- don't use `Give Up` unless there are 4 or more consecutive non-understandings, at least 30 turns in the dialog, and the ratio of non-understandings so far is above 0.8 (this strategy should be used very sparingly, only when things are going very badly).

This set of rules encapsulates prior expert knowledge. It is used to ensure that the system never takes an unreasonable action as well as to constrain the search space for the policy learning algorithm. In effect, the rules implement a heuristic strategy selection policy, which, instead of selecting one strategy, selects a set of valid strategies whenever a non-understanding occurs. Given these heuristic constraints, 4.2 non-understanding recovery strategies were available on average at any given point (the minimum was 1 and the maximum was 9.)

8.4.2.3 Features

We identified a large set of features that carry potentially relevant information for predicting the likelihood of successful for individual recovery strategies. These features can be grouped into three categories, briefly described below. The full set of features is presented in Table 44.

- **features describing the current non-understanding**. We characterized the current non-understanding in terms of a large set of features extracted from different knowledge sources in the system. These include **speech recognition features**, such as acoustic and language model scores, speech rate, signal and noise levels, clipping information; **lexical features**, such as the number of words, presence and absence of confirmation markers; **language understanding features**, such as various goodness-of-parse scores, number of grammar slots; **inter-hypothesis features** reflecting the differences in the top recognized hypotheses between the male and female recognition engines; **other features** such as timing information (e.g. barge-ins and timeouts) and the non-understanding type (e.g. no-parse vs. rejection);
- **features describing the current non-understanding segment:** the length of the current non-understanding segment; information about which recovery strategies were already taken in the current non-understanding segment, etc.;

Table 44. Features for predicting the likelihood of success for non-understanding recovery strategies

The feature types (marked in the T column) are encoded as follows: R=real, C=count, N=nominal, B=boolean
The derived features are encoded as follows:
>m = binary version indicating if the feature value is greater than the mean value of the feature in the dataset
>0 = binary version indicating if the feature value is greater than 0
>1 = binary version indicating if the feature value is greater than 0
>2 = binary version indicating if the feature value is greater than 0
>4 = binary version indicating if the feature value is greater than 0
dtf = difference between the current feature value and the feature value in the first turn in the dialog
dtp = difference between the current feature value and the feature value in the previous turn in the dialog

Feature name	Type	Derived features	Feature Description
Features describing the current non-understanding: speech recognition			
engine_id	B		the identity of the recognition engine that generated the selected hypothesis (male or female)
am_score	R	norm, >m	the acoustic model score
lm_score	R	norm, >m	the language model score
decoder_score	R	norm, >m	the decoder score
acoustic_gap	R	norm, >m	indicates the difference between the current acoustic model score and the acoustic score corresponding to an all-phone model
min_word_conf	R		the minimum word-level confidence score
avg_word_conf	R		the average word-level confidence score
max_word_conf	R		the maximum word-level confidence score
frame_num	C	>m	the number of frames
word_num	C	norm, >1, >2, >4	the number of words in the utterance
word_num_class	N		nominal feature indicating whether the utterance Rains 1 word, 2 words, 3 or 4 words, or more than 4 words
unconf_num	C	norm, >0, >1	number of unconfident words (Sphinx tags individual words as unconfident if no trigram is found in the language model ending in the current word, and a bigram back-off is forced)
unconf_ratio	R	>m	the percentage of unconfident words in the hypothesis
speak_rate	R	>m	speech rate, computed as number of frames per word
speak_rate_phones	R	>m	speech rate, computed as number of frames per phone
speak_rate_syl	R	>m	speech rate, computed as number of frames per syllable
npow	R	>m	noise level
pow	R	>m	signal level
pow_npow_diff	R	>m	signal-to-noise ratio
clip_info	C		information about whether or not the signal is clipped

Features describing the current non-understanding: lexical			
mark_confirm	B		presence of confirmation markers
mark_confirm_barge_in	B		presence of confirmation markers, and the utterance was a barge-in
unconf_mark_confirm	B		confirmation markers are present, but they are tagged as unconfident by the recognizer
mark_disconfirm	B		presence of disconfirmation markers
mark_disconfirm_barge_in	B		presence of disconfirmation markers, and the utterance was a barge-in
unconf_mark_disconfirm	B		disconfirmation markers are present, but they are tagged as unconfident by the recognizer
Features describing the current non-understanding: language understanding			
slot_num	C	>1, >2	number of grammar slots
rep_slots_num	C	>0	number of repeated grammar slots (wrt the previous turn)
new_slots_num	C	>0	number of new grammar slots (wrt the previous turn)
words_per_slot	R	>1, >2	average number of words per grammar slot
uncov_num	C	>0, >1, norm	number of words not covered by the parse
uncov_ratio	R	>m	the percentage of words not covered by the parse
frag_num	C	>1	the number of fragments in the parse
frag_ratio	R	>m	the percentage of fragments in the parse
gap_num	C	>0, >1	the number of gaps in the parse
frag_and_gap_num	C	>1	the number of fragments and gaps in the parse
hyp_num_parses	C		the number of alternative parses generated for this recognition hypothesis (due to grammar ambiguities, Phoenix can sometimes generate multiple parses for a single recognition hypothesis)
total_num_parses	C		the total number of alternative parses generated for this user input
num_parses_ratio	R		hyp_num_parses divided by total_num_parses
Features describing the current non-understanding: inter-hypotheses			
ih_diff_lexical	B		the two recognition hypotheses from the male and female recognition engine are different
ih_diff_lexical_one_word	B		the two recognition hypotheses from the male and female recognition engine are different and they both contain only one word
ih_am_score_norm_diff_to_max	R	>0	the difference between the acoustic model score of the current hypotheses to the maximum acoustic model score of the two hypotheses
ih_am_score_norm_diff_to_min	R	>0	the difference between the acoustic model score of the current hypotheses to the minimum acoustic model score of the two hypotheses
ih_lm_score_norm_diff_to_max	R	>0	the difference between the language model score of the current hypotheses to the maximum language model score of the two hypotheses
ih_lm_score_norm_diff_to_min	R	>0	the difference between the language model score of the current hypotheses to the minimum language model score of the two hypotheses
ih_frag_ratio_diff_to_max	R	>0	the difference between the fragmentation ratio of the current hypotheses to the maximum fragmentation ratio of the two hypotheses
ih_frag_ratio_diff_to_min	R	>0	the difference between the fragmentation ratio of the current hypotheses to the minimum fragmentation ratio of the two hypotheses

Features describing the current non-understanding: other			
slots_matched	C	>0, >1	the number of grammar slots that matched an open dialog expectation
slots_relevant	C	>0, >1	the number of grammar slots that are relevant to the system (this excludes for instance courtesy slots like [please])
slots_blocked	C	>0, >1	the number of grammar slots that matched a closed dialog expectation
first_level_matched	C	>0, >1	the first level in the expectation agenda where a slot from the current input matched an open expectation
last_level_matched	C	>0, >1	the last level in the expectation agenda where a slot from the current input matched an open expectation
last_level_touched	C	>0, >1	the last level in the expectation agenda where a slot from the current input matched a (open or closed) expectation
matched_in_focus	B		the input matched the dialog expectation in focus (the first level on the agenda)
barge_in	B		the user barge-in on the system
turn_number	C	>0, >1, >m	the turn number
no_parse	B		indicates that no parse was constructed for the selected recognition hypothesis
confidence	C	>.25, >.50, >.75, >m	the confidence score of the hypothesis
Features describing the current non-understanding segment			
last_turn_nonu	B		indicates if the previous turn was a non-understanding
num_prev_nonu	C	>1, >2	indicates how many consecutive non-understandings preceded the current user turn
num_prev_not_nonu	C	>1, >2, >5	indicates how many consecutive turns that were not non-understandings preceded the current turn
Features describing the current dialog state and dialog history			
dialog_state_id	B		set of binary features capturing the state the dialog manager is in (there are 2 different states in the Let's Go! Public system)
last_nonu_action_id	B		set of binary features describing the identity of the last non-understanding recovery action taken
h_ratio_nonu	C		the ratio of non-understandings so far in the dialog
h_avg_am_score_norm	C	>m	the average normalized acoustic model score so far in the dialog
h_avg_lm_score_norm	C	>m	the average normalized language model score so far in the dialog
h_avg_confidence	C	>25, >50, >75, >m	the average confidence score so far in the dialog
h_avg_gap_num	C	>m	the average value for the gap_num feature so far in the dialog
h_avg_slots_matched	C	>m	the average value for the slots_matched feature so far in the dialog
h_avg_uncov_num	C	>m	the average value for the uncov_num feature so far in the dialog

- **features describing the current dialog state and the dialog history:** we encoded the 22 dialog states in the Let's Go! Public system with a set of 22 binary variables; additionally, we computed history features such as the ratio of non-understandings, and running averages of confidence scores, goodness-of-parse scores, acoustic and language model scores, etc.

8.4.2.4 Experimental design

The online learning experiment consisted of two phases: `baseline` and `learning`.

The `baseline` phase was started on March 11th 2006, and lasted for two weeks, until March 26th, 2006. The goal in this phase was to establish the baseline performance of the system. Throughout this phase, the system randomly chose a recovery strategy whenever a non-understanding occurred, while obeying the set of constraints described in subsection 8.4.2.2. Note that, although the system chooses randomly between the available strategies, this policy is not uninformed. The constraints described in subsection 8.4.2.2 encapsulate a significant amount of expert knowledge. Coupled with the random choice, they in effect create a heuristic stochastic non-understanding recovery policy.

The `learning` phase of the experiment was started on March 26th and lasted for 6 weeks, until May 5th. Throughout this phase, we used the proposed online supervised learning method to develop a recovery policy. Each morning the data collected during the previous night was semi-automatically labeled with non-understanding recovery information: each user turn that followed a non-understanding (and was not itself a non-understanding) was inspected and manually annotated as correctly understood or not. This labeling effort took about 30 person minutes every day. The models for predicting the likelihood of success for each recovery strategy were retrained based on the data collected up to that point, and introduced in the system for the following night. Consequenly, the system learned new policies on a daily basis.

At the beginning of the `learning` phase, we realized that we had to exclude the last two strategies shown in Table 43, due to their incompatibility with our local definition of successful recovery. The first excluded strategy, `Ask Start Over`, notifies the user that a non-understanding occurred and asks if the user would like to start over. This generally elicits a yes/no type answer from the user. Although this answer might be correctly understood by the system in, a correct understanding does not really represent a successful recovery from the previous non-understanding; rather, the dialog starts again from the beginning. Similarly, when the `Give Up` strategy is engaged, the system apologizes, asks the user to call during normal business hours, and hangs up. No recovery is therefore possible in this case.

8.4.2.5 Experimental results and analysis

We first evaluated the proposed approach by computing the average non-understanding recovery rate throughout the `baseline` and `learning` periods.

The presence of the two extra strategies during the `baseline` period acts unfortunately as a confounding variable in our evaluation. Since time constraints[21] prevented us from rerunning the `baseline` phase once more without these strategies, to make the comparison fair we decided to exclude all sessions from the baseline period in which these strategies were engaged (27 out of 524). Unless otherwise mentioned, the results presented below are computed by excluding these sessions. It is important to notice that this correction in fact artificially inflates the measured system performance throughout the `baseline` period. This happens because the `Ask Start Over` and `Give Up` strategies were only available during sessions with large numbers of non-understandings – due to the nature of the heuristic constraints. `Ask Start Over` was available when the non-understanding ratio

[21] Due to other experimental demands on the Let's Go! Public system, we had only a fixed period of time allocated for experimenting with this system.

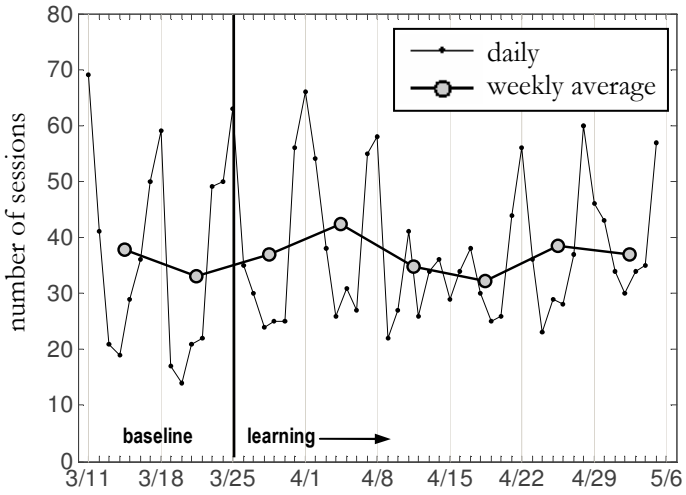


Figure 105. Number of sessions collected with the Let's Go! Public system throughout the baseline and learning phases

was above 50% and `Give Up` was available when this ratio was above 80%. By eliminating any session that contained one of these strategies, we also eliminated a significant number of other non-understandings that were not successfully recovered; the average non-understanding recovery rate in the eliminated sessions was 10.7%, as compared to 34.2% in the rest of the sessions. As a consequence, the `baseline` performance is artificially inflated. Nevertheless, as we shall see below, a learning effect is still detected.

In Figure 105 we show the number of sessions collected with the system every night. The spikes correspond to the weekends, when the number of calls to the system increases significantly. The comparatively lower number of calls for April 15th is explained by a bug that crashed the system for part of that day.

We evaluated the performance of the learned non-understanding recovery policy in terms of the average non-understanding recovery rate (ANRR). The daily average non-understanding recovery rate (throughout the `baseline` and `learning` phases) was computed as the mean of the average non-understanding recovery rate within each session for that day. The results are illustrated in Figure 106. Note that the daily average recovery rate exhibits a wide variance: this is due to the fact that the number is computed as an average over the sessions, and the number of sessions in each day is not very large. However, despite the fairly wide daily fluctuations, a comparison of average recovery performance between the first two weeks (i.e. the `baseline` period) and the last two weeks of the `learning` period reveals a statistically significant improvement: from 34.2% to 38.9%, a 13.6% relative improvement ($p=0.0375$).

This improvement represents a lower bound on the actual improvement generated by the new policy. As mentioned before, the baseline performance is artificially inflated due to the exclusion of all sessions containing an `Ask Start Over` or a `Give Up` strategy. If, instead of eliminating the whole sessions we only eliminate the turns corresponding to these strategies from the sessions where they appear, the baseline performance drops to 33.1%, which would imply a 17.4% relative improvement in the average non-understanding recovery rate.

To gain a better insight into the learning process, we fitted a learning curve to the data, described by a sigmoid:

$$\mathrm{ANNR} \leftarrow \mathrm{A} + \mathrm{B} \cdot \frac{e^{C \cdot n + D}}{1 + e^{C \cdot n + D}}$$

This curve describes the evolution of performance in a temporal learning process, where n is the number of days elapsed. Learning starts from a baseline performance level A and reaches an as-

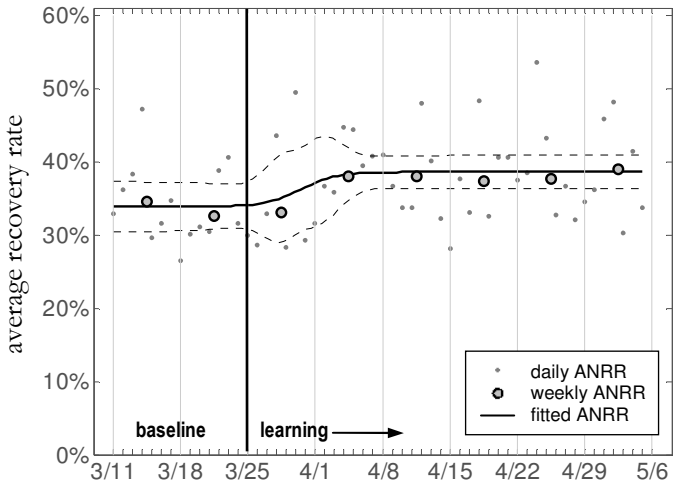


Figure 106. Improvement in average non-understanding recovery rate throughout the learning period

ymptote at A+B. The learning rate is captured by the C parameter. The D parameter allows for a shift in the starting point for the learning. We fitted this curve using a mean-squared error criterion to the daily average non-understanding recovery rates observed in our corpus. The resulting fitted curve is also illustrated in Figure 106. The resulting coefficients are: A=0.3385, B=0.0470, C=0.5566, and D=-11.44. The fitting process recovered our baseline (A=33.85%) and indicates that the asymptote ANRR is 38.55% (A+B). Furthermore, the fitting process also recovered the starting point for the learning (the curve starts moving up after Match 26th). More interestingly, the curve reveals that the system reached the asymptote performance quickly, in only ten days after the beginning of the `learning` phase.

Next, we present a more detailed analysis of the system's performance and evolution throughout the learning period. As expected, the average size of the confidence intervals for the likelihood of success prediction shrinks as more and more data becomes available – see Figure 107. This happens faster for the strategies that are engaged very often such as providing help for what users can say (`TYCS`) and slower for strategies that are engaged less often, such as asking for short answers (`ASA`).

In Figure 108.A, we illustrate (on a weekly basis) how often each strategy was engaged, as a proportion of the total number of times that strategy was available. Recall that a set of predefined rules constrained the availability of each strategy. As Figure 108.A shows, the daily invocation percentages exhibit a wide variance, in part due to data sparsity issues, in part due to the fact that the

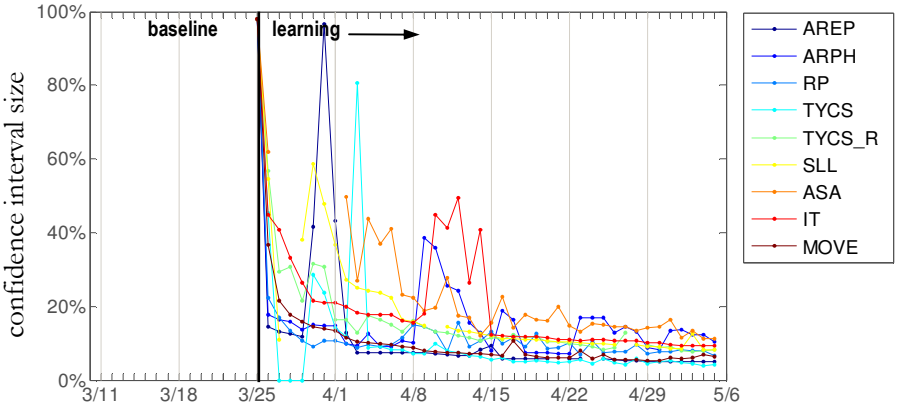


Figure 107. Average size of confidence interval for likelihood of success predictions throughout the learning phase

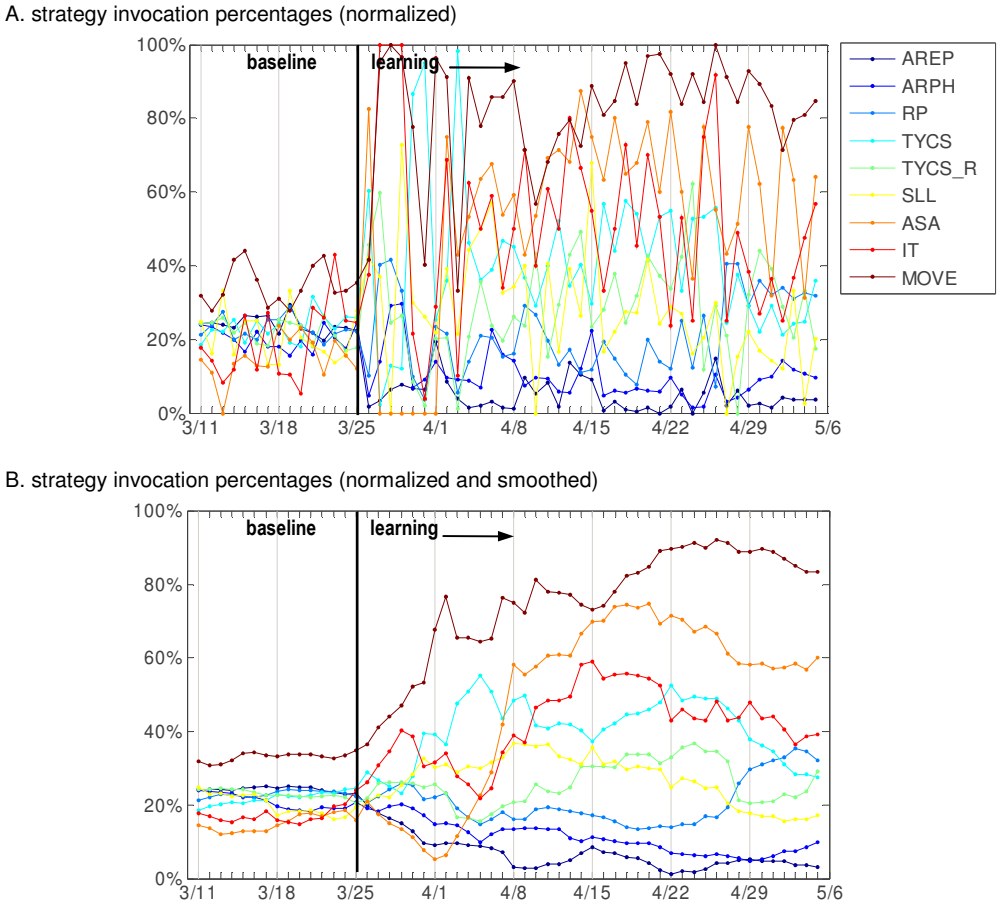


Figure 108. Normalized invocation percentages for each strategy throughout the baseline and learning phases

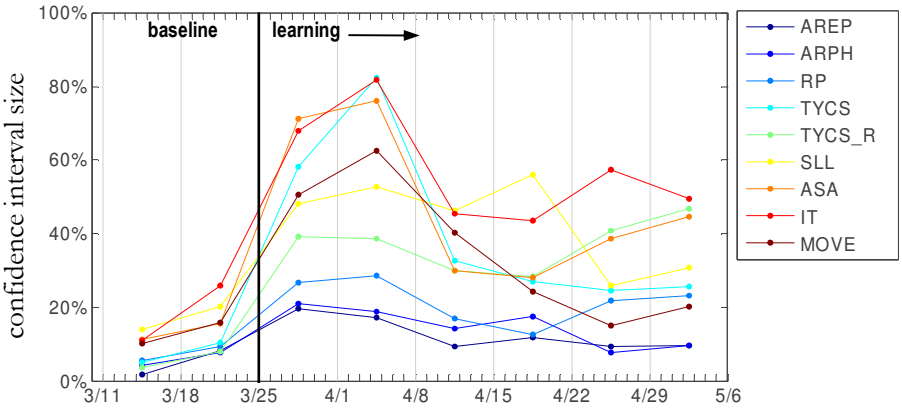


Figure 109. Volatility of normalized strategy invocation percentages

system is exploring various strategies more than others. To compensate for data sparsity, we show a smoothed version of this plot in Figure 108.B; in this case, we computed the invocation percentages by running a 7-day overlapping window over the data. As seen in Figure 108.B, before March 26th (i.e. during the baseline phase) the invocation percentages were roughly constant, because the system randomly chose between the strategies available at any given point. Once the system started learning, the invocation percentages started changing significantly. Some of the strategies, like backing off to

the neighborhood (MOVE) and providing help about what users can say (TYCS) were engaged more often than before, while others such as asking the user to repeat (AREP) or rephrase (ARPH) were engaged less often. These trends are in line with previous observations from the RoomLine data (see sub-section 8.3.3.1). Another interesting trend can be observed for the strategy that asks for shorter answers (ASA): initially, this strategy was rarely engaged (we suspect the corresponding predictor was unreliable due to small amounts of training data); as more data accumulated, the performance of this strategy was better assessed and as a consequence its usage increased significantly.

Next, we investigated to which extent the learning process has stabilized. Large oscillations in the invocation percentages for some strategies can still be observed towards the end of the learning phase. In Figure 109, we show the volatility of these invocation percentages, computed as the difference between the maximum and minimum daily invocation percentage throughout each week. As expected, during the baseline period, the volatility is low (below 20%) and is explained by random variations from one day to the next. The plot shows that throughout the first two weeks of the learning phase, the volatility is significantly increased. Afterwards it generally tends to decrease. However, it only reaches the baseline levels (i.e. below 20%) for the Ask Repeat (AREP) and Ask Rephrase (ARPH) strategies. We believe this result indicates that the policy has not yet stabilized. At the same time, Figure 106 indicates that we seem to have already reached an asymptote in terms of recovery performance. A potential explanation is that some of the strategies currently in use are equally successful at recovering from non-understandings, and, as a consequence, the highest-upper-bound selection policy creates an oscillating behavior.

We also inspected the final versions of the predictors for the likelihood of success of each individual strategy – see Table 45. Recall that these predictors are logistic regression models constructed in a supervised manner, using a large feature set in conjunction with a stepwise regression procedure. We use the Bayesian Information Criterion to regularize the stepwise regression and avoid over-fitting to the training data. For three of the strategies – asking the user to repeat (AREP), providing more interaction tips (IT), and ask the user to speak less loud (SLL) – no informative features were found. The models included only a constant factor, and thus predicted the average training set success rate for each of these strategies. A potential explanation is that the number of training samples collected by the end of the experiment for each of these strategies is still relatively low (653, 273, and 300 respectively).

Two of the models, the ones corresponding to the Ask Rephrase and Move On strategies have selected a single feature as predictive. The Ask Rephrase model uses information about whether or not the previous recovery action was Move On (last_nonu_action:MOVE). The Move On model uses information about the number of words in the recognized hypothesis: the larger the number of words, the more likely it is that this strategy will succeed. The remaining four models, for the strategies that asked the user to provide a shorter answer (ASA, 637 samples), repeating the system prompt (RP, 2532s), and the two help strategies (TYCS, 3698s; TYCS_R, 989s) are based on at least four predictive features. Some of the most informative features were the dialog state indicators: for instance the two help strategies are likely to be more successful during explicit confirmation states. This result corresponds to our intuition: for explicit confirmation states the help strategies instruct the users to provide a simple yes or no answer, and this generally leads to successful recovery. Dialog history features are also informative: the likelihood of recovery is proportional with how well the dialog went so far. In particular, for the Repeat Prompt strategy, the likelihood of success decreases as the ratio of non-understandings encountered previously in the dialog (h_ratio_nonu) increases. Similarly, for the (Terse)You-Can-Say and Restart strategy, the likelihood of success increases as the average number of slots matched in the dialog (h_avg_slots_matched – a measure of the discourse understanding quality) increases.

Finally, it is important to notice that the individual performance of the likelihood-of-success predictors for each individual strategy is not very high. In fact, for three of these strategies no informative features were found and the models predicted the majority baseline. For the other strategies, the models showed only small relative improvements in Brier score over a majority baseline (5-12% relative.) Nevertheless, improvements in recovery performance were possible. Better individual pre-

Ask Rephrase (ARPH)		
Feature	Coef.	Effect
k	-2.16	-
last_nonu_action:MOVE	1.18	+

Ask Short Answer (ASA)		
Feature	Coef.	Effect
k	-2.65	-
num_prev_not_nonu>2	1.42	+
total_num_parses	0.60	+
dialog_state_id:RequestDepartureInNeighborhood	1.73	+
avg_word_conf	4.98	+

Move On (MOVE)		
Feature	Coef.	Effect
K	-0.86	-
word_num	0.06	+

Repeat Prompt (RP)		
Feature	Coef.	Effect
k	-0.22	-
dialog_state_id:RequestArrivalPlace	-0.62	-
dialog_state_id:GetDepartureNeighborhood	-1.04	-
h_ratio_nonu	-1.96	-
dialog_state_id:ExplicitConfirm(departure_place)	1.37	+
dialog_state_id:ExplicitConfirm(time)	1.65	+
dialog_state_id:RequestTravelTime	0.89	+
dialog_state_id:ExplicitConfirm(arrival_place)	1.32	+
dialog_state_id:ExplicitConfirm(neighborhood)	1.62	+

(Terse)You-Can-Say (TYCS)		
Feature	Coef.	Effect
k	0.11	+
h_ratio_nonu	-2.28	-
dialog_state_id:ExplicitConfirm(route_number)	1.49	+
dialog_state_id:ExplicitConfirm(departure_place)	1.39	+
dialog_state_id:ExplicitConfirm(arrival_place)	1.54	+
dialog_state_id:ExplicitConfirm(time)	1.48	+
dialog_state_id:ExplicitConfirm(neighborhood)	1.90	+
dialog_state_id:ExplicitConfirm(uncovered_neigh)	2.22	+

(Terse)You-Can-Say and Restart (TYCS_R)		
Feature	Coef.	Effect
k	-3.64	-
h_avg_slots_matched	3.65	+
dialog_state_id:ExplicitConfirm(time)	1.67	+
dialog_state_id:ExplicitConfirm(departure_place)	1.35	+
dialog_state_id:RequestDepartureInNeighborhood	-1.38	-

Table 45. Models for predicting likelihood of success for individual recovery strategies at the end of the learning period

dictors can lead to further performance improvements. Based on our experience, we believe that two aspects will play a key role: (1) identifying other informative features, and (2) further extending the existing set of non-understanding recovery strategies.

8.4.3 Concluding remarks

In this section (8.4), we have proposed and evaluated a data-driven approach for developing non-understanding recovery policies over large sets of recovery strategies.

The approach has a number of advantages over current heuristic solutions. First, it is data-driven and online in nature. The system starts from an agnostic policy and learns through experience how to better engage the various recovery strategies. The starting point need not be completely ag-

nostic: the approach allows us to encode certain expert knowledge as constraints on the policy space. While learning, the proposed approach relies on the highest-upper selection method to strike a balance between exploiting strategies that are known to be good and gaining more knowledge about the other, less well understood strategies. In the process, the policy is adapted to the characteristics of the domain in which the system operates. Our initial experiments with a publicly available spoken dialog system indicate that the proposed approach leads to statistically significant improvements (in terms of average non-understanding recovery rate) over a heuristic policy designed by a domain expert. In our case, the system learned a better recovery policy over a set of nine non-understanding recovery strategies in a fairly short time period: 10 days, or about 500 dialog sessions.

Another advantage of the proposed approach is that it scales well, as the number of non-understanding recovery strategies increases. In fact, the approach would in principle allow us to add new strategies into the mix later on. The highest-upper-bound policy will explore and exploit them accordingly. The good scalability property stems from the fact that the learning happens independently for each strategy and is local in nature: the learning process focuses on the next, immediate user turn, rather than on a global dialog optimization. The assumption we are making is that non-understanding recovery strategies have only local effects, and no long-term interactions exist. In general, this assumption does not hold for all recovery strategies and domains; for instance, we have seen that we had to eliminate the `Ask Start Over` and `Give Up` strategies due to their incompatibility with the local definition of success. Assessing the performance of the `Move On` strategy just in terms of recovery rate is also problematic (this latter problem could be addressed by using a more refined metric, such as the ones described in subsection 8.3.3.1.) However, with a carefully chosen set of recovery strategies, we believe the costs of violating this assumption are surpassed by the benefits it affords: good local recovery performance does generally sum up to good global dialog performance.

The proposed approach, as described and evaluated in this section, is supervised in nature. The system learns predictors for the likelihood of success for each recovery strategy in a supervised fashion: throughout the experiment, a human annotator manually labeled the instances in which the system successfully recovered, and the ones in which it failed to do so. This is generally a costly endeavor, and it can be regarded as a drawback of the proposed approach. We believe however that the implicitly-supervised learning paradigm introduced earlier in Chapter 5 could be applied to this problem, and might lead to performance improvements similar to the fully-supervised approach. Instead of defining successful recovery as "the next turn is correctly understood by the system" (which requires manual labeling), we could define it as "the next turn is not another non-understanding). Another alternative is "the next turn has a high confidence score", because presumably confidence scores can provide a finer-grained assessment. A third alternative would be to use explicit confirmations: after each apparent non-understanding recovery (e.g. after each turn following a non-understanding that is not itself a non-understanding), the system would explicitly confirm the user response. As we have seen in the confidence annotation work, this interaction pattern can provide meaningful labels at a sufficiently high level of accuracy. Although these labels would not be perfect, we believe the potential benefits (i.e. an unsupervised online approach for tuning recovery policies) make this a line of research worth exploring.

8.5 Summary and future directions

In this chapter, we have focused our attention on non-understanding recovery strategies and policies.

In an effort to add to a growing body of knowledge regarding the properties of various non-understanding recovery strategies, we performed an empirical investigation of ten such strategies in the context of a telephone-based, mixed-initiative spoken dialog system. We introduced four different metrics for assessing recovery performance, and performed a comparative analysis of these strategies under two different conditions: (1) when engaged by an uninformed policy and (2) when engaged by a human operator in a wizard-of-oz setup.

The results we obtained corroborate in part previous results reported in the literature: the high performance of the `Move On` strategy (i.e. ignoring the problem and continuing with an alterna-

tive dialog plan) is consistent with previous experiments by Skantze [115]; the distribution of user response types after non-understanding recovery strategies is similar to the one observed by Shin et al. [113], and Choularton and Dale [26]; the observation that users are generally more likely to rephrase than to repeat (regardless of the recovery strategy) confirms a previous study by Goldberg [39]. At the same time, it is important to notice that observations regarding the relative performance of various strategies and user responses do not always generalize well across domains. For instance, we found conflicting evidence with Goldberg's study [39] regarding the relative likelihood of recovery for user repeats and user rephrases. In addition, we also found that the success of a non-understanding recovery strategy, and in general global dialog performance, can be strongly influenced by the recovery policy used to engage the strategy. Other factors, such as the user population, environmental conditions, etc. can also play an important role.

These results indicate that fixed rules-of-thumb (even based on quantitative, empirical observations) are not always reliable. Instead, spoken dialog systems should learn from their interactions throughout their lifetime and continuously adapt their policies to the particular environments in which they operate. With that in mind, we have proposed an on-line, data-driven solution for developing non-understanding recovery policies over large sets of recovery strategies. In the proposed approach the system constructs estimators for the likelihood of success for each recovery strategy (together with confidence bounds for these estimators), and uses a highest-upper-bound selection method to balance between exploration and exploitation. Initial experiments with a publicly available spoken dialog system indicate that the system is able to learn in a relatively short time period a policy that performs better than a heuristic, expert-designed policy.

The work described in this chapter has also opened up a number of interesting avenues for future work. First, the high performance of the `Move On` strategy, together with prior evidence from Skantze [115], points to a road less traveled in spoken dialog systems: instead of trying to recover from errors, ignore them and try an alternative dialog plan. Currently, this strategy is used only at certain points in the dialog, pre-specified by the system author. In addition, when the strategy is engaged, the dialog engine marks the current request as failed and moves on to the next action: it is the author's responsibility that the dialog plan can continue and be completed successfully despite this local failure. The system never returns to the failed request. In future work, it would be interesting to explore in more detail potential uses and extensions of this strategy, as well as its drawbacks. For instance, it would be interesting to identify more situations in which the `Move On` strategy is applicable. Could the dialog engine automatically infer from a given dialog plan when this strategy is applicable (and therefore remove the burden from the system's author)? We believe different solutions to this problem might exist in the context of a plan-based dialog manager like RavenClaw.: the system could continue with the next question, mark the request that triggered the non-understanding as "temporarily failed" and return to it at a later point in the conversation; with some hints from the system's author, it could probably also do this at the level of dialog plans rather than simple requests. Building a policy that uses such a strategy does raise some interesting challenges since long-term effects come into play: abandoning a dialog plan and switching to an alternative one generally means the losing information and the time invested in the current plan so far.

The proposed online method for learning non-understanding recovery policies only constitutes a first step towards building adaptive, self-improving systems. Much remains to be done. For instance, it would be interesting to investigate different implicitly supervised solutions in an effort to completely eliminate the developer from the loop. Longer longitudinal studies would be necessary to better understand whether or not the policy stabilizes in time, and how the proposed online learning paradigm handles shifts in the environmental conditions. What happens if we change the language model and in the process improve speech recognition performance by 20% relative? What happens if we add three new strategies and 50 more features in the middle of the learning phase? In addition, it would be interesting to understand how well the proposed solutions scale to very large numbers of strategies. Prompt design is essential in developing effective non-understanding recovery strategies. Using the proposed policy learning approach, we could introduce 5 prompt variants for each strategy (therefore creating a set of 5x10=50 strategies), and let the system learn which of these variants is

most appropriate. Finally, it would be interesting to investigate whether or not the learning paradigm could be adjusted to deal with a non-local definition of successful recovery and therefore handle more complex recovery strategies with longer term effects.

PART V.

CONCLUSION

Chapter 9

Conclusion

The work described in this dissertation aims to create the mechanisms for seamlessly and efficiently recovering from understanding-errors in conversational spoken language interfaces. In this final chapter we briefly recap the motivation for this work and summarize our accomplishments. We draw a number of conclusions based on the work conducted and the results obtained. Finally, we discuss a number of questions that remain unanswered and directions for further extending this work.

Spoken language interfaces hold a number of great promises. They rely on natural language and therefore require little or no user training; this in turn makes them accessible to a large user population. Additionally, natural language allows us to express complex concepts and construct complicated queries with little effort, which creates an opportunity for a very efficient interaction. Also, speech is an ambient rather than attentional modality. This makes it appropriate in eyes-busy and hand-busy situations; this flexibility means that ultimately speech technologies could play a key enabling role in the quest for ubiquitous computing and building ambient intelligences.

Unfortunately, despite these great promises and many years of progress in the science and technology of the underlying components, today's spoken language interfaces are still very brittle when confronted with understanding-errors. This lack of robustness seriously affects the function of current spoken language interfaces, often leading to complete conversational breakdowns, task failure, and increased user frustration. Anecdotal evidence, as well as statistics reported in the literature [62, 68, 89, 121] indicate that conversational speech-based interfaces are oftentimes seen as annoyance rather than the enabling technology we would like them to be.

Most understanding-errors stem from current limitations in speech recognition technology. As a consequence they appear in all domains and interaction-types. Automated speech recognition is a difficult task to begin with, and, in the context of conversational spoken language interfaces, the difficulties are further exacerbated by the conditions under which these systems are meant to operate. Generally, spoken language interfaces are targeted at wide user populations and therefore they have to accommodate large variations in speaking styles (e.g. native, non-native, various accents) and quality of the input lines (e.g. land lines, cell phones, VoIP, PDA microphones). If we add to the mix the disfluencies that characterize conversational speech, it is not surprising that recognition error rates range between 15 and 40% (and even higher) in any but the simplest systems.

Left unchecked, speech recognition errors propagate to the higher levels of these systems where they can lead to two types of understanding-errors: **misunderstandings** or **non-understandings**. In a misunderstanding, the system incorrectly understands the user: for instance,

the user says *"Boston"* but the system understands *"Austin"*. In a non-understanding, the system does not understand the user at all: for instance, the user says *"Urbana Champaign"*, but the speech recognition engine produces *"OKAY IN THAT SAME PAY"* - which makes no sense semantically in the current dialog context. Repeated studies have shown that these errors exert a significant negative impact on the overall quality and success of the interactions [128, 129].

Two approaches to increasing robustness can be envisioned: (1) prevent the errors altogether (e.g. build better speech recognition and spoken language understanding technologies) and (2) work under the assumption that some errors will always be there and build the capabilities for recovering from them through conversation, by interacting with the user. Of course, these two methods do not stand in opposition; rather, a combined effort would lead to the best results.

The work described in this dissertation focuses on the second approach: it aims to create the mechanisms for seamlessly and efficiently recovering from errors through conversation. We have argued that three key capabilities are needed to accomplish this goal. First, systems must be able to accurately **detect errors**, preferably as soon as they happen. Second, systems should be equipped with a rich repertoire of **error recovery strategies** that can be used to set the conversation back on track (e.g. asking the user to repeat, to rephrase, to speak softer or louder, providing help, confirming a piece of information, etc.). Third, systems should be able to choose wisely between the available recovery strategies at runtime (when should a system ask the user to repeat? when should it ask the user to rephrase? when should it provide more help?, etc.). In other words, spoken dialog systems need good **error recovery policies** to guide their error recovery actions.

These three issues, i.e. error detection, error recovery strategies and error recovery policies, together with the two types of understanding-errors introduced before, define the coordinates for the long-term research program we have articulated in this dissertation (see Figure 110). Within this space, a number of important and interesting research problems can be identified. At the error detection level, we are mostly faced with a pattern recognition problem. At the recovery strategies level we are faced with a human-computer interaction design problem. Last, at the recovery policy level we are faced with a problem of control under uncertainty.

In this work, the context for addressing these problems has been practical, real-world spoken dialog systems. These are complex, layered systems that subsume multiple components and interact with a dynamic world, more specifically with another intelligent, goal-driven individual. Their complexity represents both a challenge and an opportunity. The different components in a spoken language interface (e.g. speech recognition, language understanding, dialog management, etc.) manipulate different types of knowledge. We have seen throughout this work that features extracted from these different knowledge sources can provide useful, orthogonal information for solving prob-

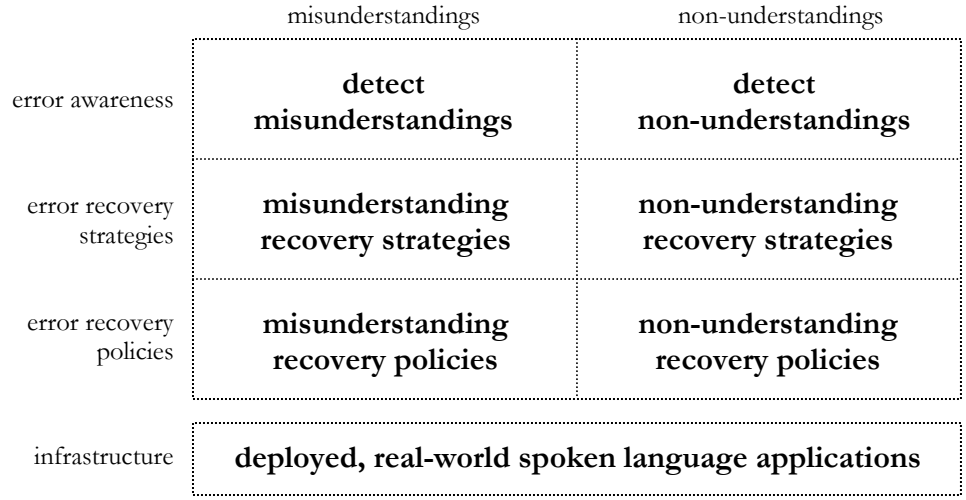


Figure 110. A research program for increased robustness in conversational spoken language interfaces

lems. In addition, the presence of an intelligent, goal-oriented, and invested individual "at the other end of the line" creates interesting learning opportunities. We have seen that an implicitly-supervised learning paradigm can enable significant autonomous, online learning and pave the way towards self-improving systems.

At the same time, the error detection and recovery problems and solutions that we have discussed in this work are not confined just to spoken dialog systems. Error handling is an important issue in general in interactive systems of any type. For instance, many similarities can be drawn between conversational spoken language interfaces and robotics. In both cases, we are dealing complex, layered systems that interact with a dynamic world. In both cases the systems have limited sensing abilities. As expected, cross-fertilization between these areas has happened in the past, and we believe will continue to happen in the future. We conjecture that some of the solutions proposed in this dissertation, such as expectation-driven error detection, belief updating using compressed representations, learning from implicit supervision, can generalized and applied in other domains beyond conversational spoken language interfaces.

9.1 Summary of contributions

The dissertation work described in this document brings a number of contributions within the error handling problem space outlined above. At the same time, this work does not complete the entire research program we have articulated. Rather, it is best viewed as a concerted effort at advancing the state-of-the-art in a number of these areas, while also raising a number of other interesting scientific and technical questions.

The main contributions of this work are summarized in Figure 111. In this section, we briefly review these contributions and present several concluding remarks. In the next section, 9.2, we discuss a number of remaining open questions and we outline a few directions for future research within this problem space.

We began in Chapter 2 by discussing in detail the two types of understanding-errors that commonly affect conversational spoken language interfaces: misunderstandings and non-understandings. To identify the sources of these errors, we proposed a four-level model of error-source analysis anchored in Clark's model of grounding in communication [27]. An analysis of corpora from two different dialog domains confirmed that most understanding-errors do stem from the speech recognition level. At the same time, language-domain errors, such as out-of-grammar, out-of-domain and out-of-application-scope utterances, also contribute to increasing the total number of understanding-errors. Both misunderstandings and non-understanding exert a significant negative impact on overall system performance. We proposed a data-driven approach for quantifying this impact. The central idea is to relate the total number of misunderstandings and non-understandings in a dialog session to the overall performance in that session by using a regression model. The proposed models corroborate previous intuitions and can help quantify and shed more light on the impact of understanding-errors on performance. For instance, experiments in two different dialog domains revealed that the impact of misunderstandings and non-understanding on performance is non-linear and the relative costs of these errors are different across domains.

Next, in Part II (Chapter 3 and Chapter 4) we outlined the experimental platform and infrastructure we have developed to support the proposed error handling research program. The infrastructure consists of **RavenClaw, a plan-based, task-independent dialog management framework (C9)** and a number of spoken dialog systems that have been developed within that framework. In Chapter 3 we provided an overview of the RavenClaw framework: we outlined the architecture, functionality, core algorithms in the RavenClaw dialog engine, as well as the set of available task-independent conversational strategies. The key characteristic of the RavenClaw dialog management architecture is that it enforces a clear separation between the domain-specific and domain-independent aspects of the dialog control logic. The domain-specific aspects are provided by the system author via a dialog task specification, essentially a hierarchical plan for the interaction. The dialog engine manages the conversation based on the dialog task specification, and in parallel automatically

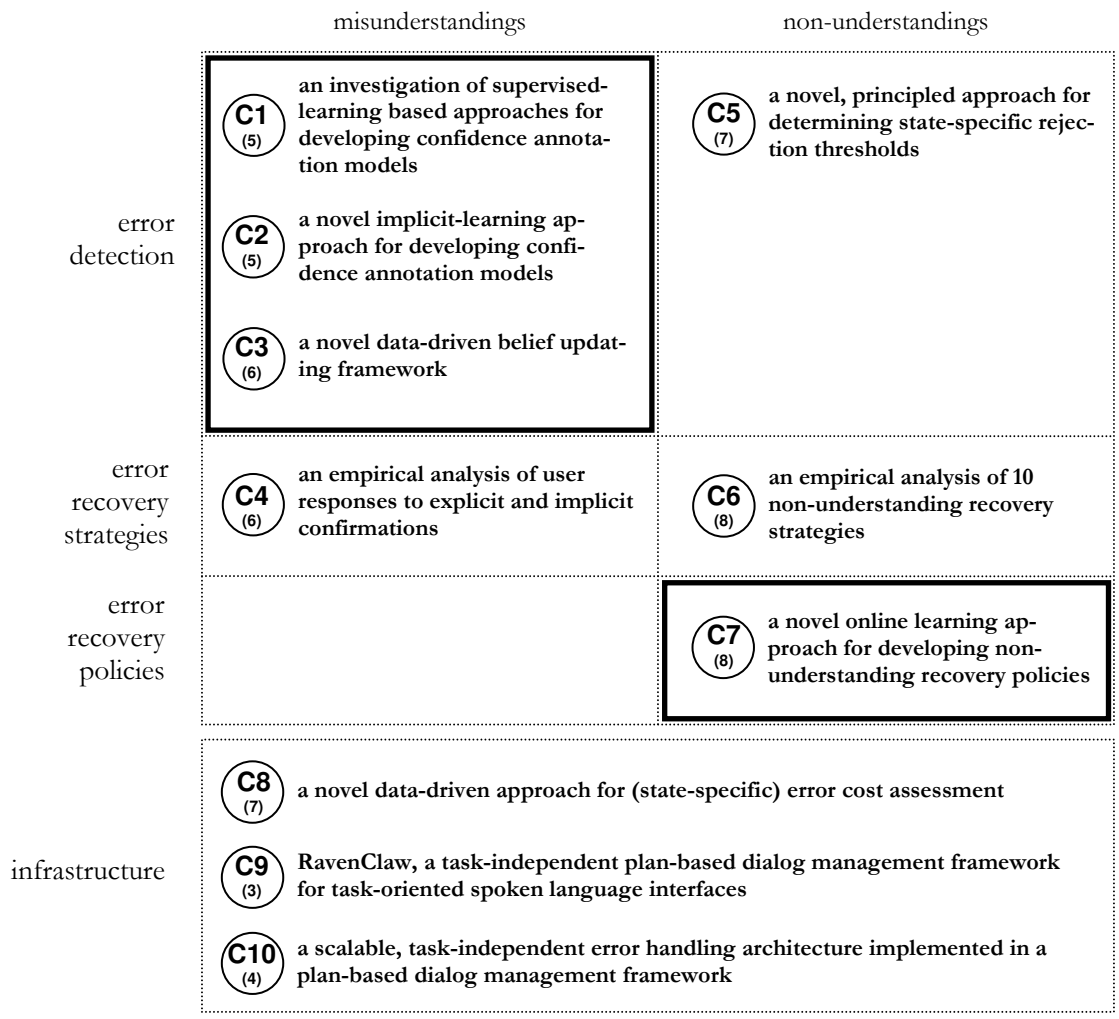


Figure 111. A summary of contributions; for indexing purposes, the number in parentheses indicates the chapter in which each contribution is described

ensures a basic set of conversational skills such as error handling, timing and turn-taking. This decoupling significantly lessens the system development effort, promotes both portability and reusability and ensures uniformity and consistency in behaviors within and across systems. To date, the RavenClaw dialog management framework has been used to develop and successfully deploy a number of spoken dialog systems operating in different domains and interaction types.

In Chapter 4 we described the **error handling architecture in the RavenClaw dialog management framework (C10)**. The error handling architecture is task-independent: it decouples both the error handling strategies and the error handling decision process from the actual dialog task specification. This decoupling significantly simplifies the authoring effort. System developers describe the dialog task control logic under the assumption that inputs to the system will always be perfect. The responsibility for ensuring that the system operates with correct information and that the dialog progresses normally towards its goals is delegated to the error handling decision process in the dialog engine. Whenever necessary, this process will automatically engage various recovery strategies to set the conversation back on track. The task-decoupled implementation favors portability of the recovery strategies and policies across domains, and ensures consistent error handling behaviors within and across systems. Together with the surrounding RavenClaw dialog management framework and the systems developed within this framework, the error handling architecture provides the infra-

structure for the rest of the research program we have tackled in this dissertation.

Part III of this dissertation (Chapter 5 and Chapter 6) is dedicated to the issue of handling misunderstandings. This part is largely focused on developing better misunderstanding detection mechanisms. In addition, we have also conducted an empirical investigation of two misunderstanding recovery strategies: explicit and implicit confirmation. In Chapter 5 we discussed the problem of semantic confidence annotation, or utterance-level detection of misunderstandings. We opened the chapter with **an in-depth investigation of four supervised learning techniques for building confidence annotation models using data from three different dialog domains (C1)**. We investigated a number of issues that have received less attention in the literature, such as evaluation metrics for confidence annotation, sample efficiency of different supervised learning techniques and how well the confidence annotation models generalize across domains. Our results highlight the importance of using proper probabilistic scoring rules that measure both accuracy and calibration (e.g. log-loss, Brier score) when evaluating confidence annotation models for conversational spoken language interfaces. In addition, the results show that some classifiers (e.g. AdaBoost, Naïve Bayes) suffer from a lack of calibration; this can be corrected in a simple, post-training calibration step. Although similar performance is attained by supervised learning techniques (i.e. logistic regression, decision trees, AdaBoost), the logistic regression models are the most sample efficient. When the amount of available training data is small, these models significantly outperform the others. Finally, we have investigated how well confidence annotation models generalize across different dialog domains. Results indicate that, although some models generalize well across domains, this is not always the case. Furthermore the transfer can sometimes be asymmetric. We have shown that a simple post-transfer calibration procedure that relies on a small number of labeled data-points in the target domain can generally improve the performance of the transferred model.

In the second part of Chapter 5 we proposed **a novel, implicitly-supervised learning paradigm for building confidence annotation models (C2)**. The proposed approach eliminates the need for a pre-existing corpus of labeled utterances. Instead, the system obtains the labels directly from the interaction, from user corrections following the system's explicit confirmation actions. Batch experiments conducted with data from two deployed spoken dialog systems show that the proposed approach can attain around 80% of the performance of a fully-supervised model. Furthermore, the proposed implicitly-supervised paradigm favors an online approach to learning confidence annotation models. A spoken dialog system could start with an agnostic confidence annotation model and aggressively confirm every piece of information received from the user. As the system collects more data and builds a more reliable confidence annotation model, it can start trusting it more and relying less on explicit confirmations. We believe the proposed implicitly-supervised learning approach can be applied in a number of other learning problems in spoken dialog systems and in general in other interactive systems. We conjecture that this paradigm can enable significant autonomous learning and represents an important step towards building continuously self-improving systems.

While confidence scores can be used to perform turn-level detection of misunderstandings, ideally spoken dialog systems should fuse evidence from multiple turns in the dialog to continuously monitor and improve the accuracy of their beliefs. In Chapter 6, we proposed **a scalable, data-driven approach for updating beliefs in spoken dialog systems (C3)**. The proposed approach relies on a compressed representation of beliefs and casts the 1-step belief updating problem as a multinomial regression task: given an initial belief over a concept, a system action with respect to that concept, and a follow-up user response (as this response is perceived by the system), construct an updated belief in a manner as accurate as possible. The approach bridges ideas from previous work on confidence annotation and correction detection; it provides a unified framework for integrating evidence collected from multiple turns in a conversation to continuously monitor and improve the accuracy of the system's beliefs. Empirical results show that the proposed approach constructs beliefs that are significantly more accurate than previous heuristic solutions and produces significant gains in both the efficiency and the effectiveness of the interaction. Independent of these performance gains, the approach has a number of other good properties: it scales well with the number of

concepts in the dialog and their cardinality, it is sample efficient and portable.

To better understand the challenges we face in the belief updating task, we have conducted **an empirical investigation of user responses to explicit and implicit confirmations (C4)**. The results, also described in Chapter 6, corroborate prior observations by Krahmer et al. [63], and indicate that user responses to confirmation actions cover a wide language spectrum, especially after implicit confirmations, and especially if the information to be confirmed is incorrect. Secondly, we also found that users interact strategically with the system: oftentimes, they correct the system only if the correction is essential for accomplishing the task at hand.

Part IV of this dissertation (Chapter 7 and Chapter 8) is dedicated to the issue of handling non-understandings. In Chapter 7 we focused on the issue of detecting non-understandings, more specifically rejection non-understandings. Spoken dialog systems often reject utterances if the confidence score falls below a preset rejection threshold; in effect, the system will create a rejection non-understanding to avoid a potential misunderstanding. In Chapter 7 we proposed **a principled method for determining state-specific rejection thresholds (C5)**. The approach relies on the data-driven error cost assessment models we introduced earlier, in Chapter 2. We extended these models to account for state distinctions and used them to **infer state-specific costs for various types of errors (C8)** involved in the rejection trade-off, like false-rejections and false-acceptances. The resulting costs, based on data from a deployed spoken dialog system, corroborate our intuitions and a number of other anecdotal observations made throughout the use of this system. The inferred costs allow us to optimize rejection thresholds in a state-specific manner.

In Chapter 8 we turned our attention to the set of non-understanding recovery strategies and the problem of constructing non-understanding recovery policies. In the first part of this chapter, we reported on **an in-depth investigation of 10 non-understanding recovery strategies (C6)**. In an effort to add to an existing body of knowledge regarding the relative advantages and disadvantages these strategies, we performed a comparative analysis of these strategies, under two different conditions: when engaged in an uninformed manner, and when engaged using a "smarter" policy implemented by a human in a wizard-of-oz setup. We analyzed recovery performance, the distribution of user responses to these strategies, and which user responses lead more often to successful recovery. To some extent, our results corroborate observations previously made by others in different domains (e.g. the success of `Move On` and `Help` strategies, the overall distribution of user response types throughout non-understanding segments). At the same time, we also found conflicting evidence. Overall, the lesson learned is that the success of various recovery strategies is context-sensitive; factors such as the nature of the dialog task, the user population and the policy used to engage the strategies can significantly affect their performance. The solution to successful error recovery therefore lies in creating mechanisms that allow systems to adapt their behaviors to the particular characteristics of the domain in which they operate.

To this end, in the second half of Chapter 8 we have proposed **an online data-driven method for learning non-understanding recovery policies over a large set of recovery strategies (C7)**. The approach works in two steps: at each time point (say every day), we construct runtime estimates for the likelihood of success for each non-understanding recovery strategy, together with confidence bounds for these estimates. The predictors are constructed based on a large number of features extracted from different knowledge sources in the system, and data collected with the system up to that point. The confidence bounds reflect the uncertainty in each prediction, due to data sparsity. We then use a highest-upper-bound selection method in conjunction with these predictors to select which strategy to engage in at runtime. In doing so, we guide exploration while at the same time exploiting well-performing strategies. An empirical evaluation in a deployed spoken dialog system shows that the proposed approach leads to statistically significant improvements in the non-understanding recovery rate.

9.2 Concluding remarks and future work

The work described in this document is best viewed as a concerted effort contributing to the area of error recovery in conversational spoken language interfaces. The work was conducted within the confines of a research program that centers on three important components of error handling: detection, strategies and policy. However, this research program is by no means complete. Each of the work items described in this dissertation leaves specific follow-up questions (we have signaled most of them in the corresponding chapters.) In addition, this work also raises a number of more general, cross-cutting open questions. In this section, we draw a number of high-level concluding remarks and outline several interesting directions for future work.

§ Error detection

Error detection can be viewed a pattern recognition problem. A key step in solving this problem is the identification of features that can inform the error detection task. In the confidence annotation work described in Chapter 5, we showed that features from different knowledge sources in the system (e.g. speech recognition, prosody, lexical, grammar, dialog) can provide useful and complementary information for detecting errors. In the belief updating work from Chapter 6, we showed how fusing evidence from multiple turns in the conversation can give an additional boost in performance. Again high-level, domain-specific information such as priors, confusability, concept identity, played a key role in constructing more accurate beliefs. In future work it would be interesting to identify and investigate other knowledge sources that can provide additional information for this task (e.g. domain-specific knowledge, inter-concept dependencies, etc.) Another interesting direction for future research is the development of methods for automatically generalizing error detection models across domains. Training new error detection models for each new domain is a labor-intensive process that we would like to shortcut. In Chapter 5, we have performed a preliminary investigation of cross-domain generalization for confidence annotation models. The results are promising, but more research is needed. Can we do the same for the belief updating models? Can we identify the conditions under which we can expect an existing error detection model to generalize well to a new domain? Better yet, can we create the mechanisms for adapting existing error detection models to new domains in the absence of any labeled data?

§ Error recovery strategies

At the error recovery strategies level, we are mostly faced with a design or human-computer interaction issue. Our empirical investigations of confirmation strategies (subsection 6.4.3 from Chapter 6) and non-understanding recovery strategies (section 8.3 from Chapter 8) have added to an already existent but sometimes inconsistent body of knowledge regarding the function and performance of these strategies. To some degree, the results we have found are in line with previous observations. At the same time, we have also found evidence that contradicts previously reported results.

The most important lesson learned is perhaps obvious in hindsight. A consistent image about the performance of these strategies is hard to establish because their function and performance are strongly influenced by a number of contextual factors. These include the nature of the system's task, the user population, the environmental conditions, and the policy used by the system to engage these strategies. To a large extent, the solution for successful error recovery lies therefore not only in endowing spoken dialog systems with a rich set of error recovery strategies, but also in creating the mechanisms for adapting error handling behaviors to the characteristics of the environments in which the system operates.

§ Error recovery policies

Finally, at the policy level, we are faced with a problem of control under uncertainty. The work described in this dissertation focused mostly on policies for recovering from non-understandings. In section 8.4 from Chapter 8 we proposed an online approach for learning such policies from data. Experiments with a deployed spoken dialog system showed that the proposed approach leads to improvements in the non-understanding recovery rate. At the same time, a number of questions regard-

ing this approach remain open: do the improvements in recovery rate translate into overall improvements in dialog performance? Does the learned policy stabilize in time? How robust is the learning process under changes the set of recovery strategies, predictive features, or in general, environmental conditions? How does the proposed approach scale to larger numbers of recovery strategies? Can the learning paradigm be adjusted to handle a non-local definition of successful recovery and therefore accommodate more complex recovery strategies with longer term effects?

Although this dissertation does not directly contribute to the problem of developing policies for recovering from misunderstandings, it does point to an interesting approach in this area. In this work, we used the classical threshold-on-confidence model for engaging the various misunderstanding recovery strategies (explicit and implicit confirmation.) At the same time, we argued that the data-driven error-cost assessment methodology described in subsection 2.3.1 from Chapter 2 and section 7.4 from Chapter 7 can be extended to infer the costs of various confirmation actions. These costs could be used to compute confidence thresholds in a more principled manner, and optimize the misunderstanding recovery policy. This approach remains to be validated empirically.

§ Other observations and future work

Throughout this work, we have sought task-independent, scalable and adaptive solutions for various problems related to error handling. The underlying design of the error handling architecture in the RavenClaw dialog management framework confers the desired task-indepedence and scalability properties. The error handling architecture decouples the error detection mechanisms, the error recovery strategies and the error recovery policies from the dialog task and from each other. Furthermore, error handling decisions are made independently with respect to various dialog entities, i.e. concepts and request-agents. We made a tacit assumption throughout this work: error handling can be modeled as a local process, and decoupled from the actual task that the system operates with. We showed that this decoupling brings a number of important benefits: it lessens the system development effort, it promotes portability and reusability, it ensures consistency in behavior both within and across tasks, it supports dynamic generation of dialog tasks, it confers good scalability properties, and it enables learning-based approaches.

At the same time, the distributed and task-decoupled approach also has a number of limitations and drawbacks. For instance, domain-specific inter-concept dependencies are not modeled. Knowing that the start-time for a conference room reservation is 4 p.m. imposes constraints on what the end-time can be. This information can be very useful in detecting potential misunderstandings for the end-time concept. However, the current architecture treats each concept independently and does not directly take advantage of such information. In addition, complex combinations of recovery actions are not supported. For instance, in the current architecture the system cannot explicitly confirm two concepts in a single turn, e.g. "Did you say tomorrow at 5?" Furthermore, long-range effects of recovery actions are not directly modeled. To some extent, these limitations can be addressed in the context of the proposed architecture by adding global information in the local error handling decision processes. In fact, we have done so already in work reported in this dissertation: for instance, the belief updating models described in Chapter 6 make use of both domain-specific (e.g. priors, confusability, concept identity, dialog-state, etc.) and global (e.g. history of confidence scores, etc.) information; the same holds true for the non-understanding recovery policies described in Chapter 8.

Other approaches discussed in the literature, most notably reinforcement-learning [67, 82, 106-108, 114], do not make this assumption and aim to optimize the system's behavior globally. Although these approaches stand on a more theoretically sound basis, they do not scale well and are currently still impractical in real-world spoken dialog systems (for an in-depth discussion of strengths and weaknesses, see [82].) In constrast, the work described in this dissertation regards error handling as a mostly local problem. The view taken in this work is that this assumption can be safely made in a large class of task-oriented systems, and that the overall benefits outweight the potential drawbacks. To give a concrete example, we empirically showed in the belief updating work described in Chapter 6 that good local decisions do sum up to significant improvements in overall dialog performance. Perhaps further performance improvements would be possible by taking into account other inter-

concept dependencies and long-range effects. In future work, it would be interesting to investigate more closely the impact of this assumption on the proposed solutions, both in terms of scalability and performance gains.

The third desirable property we sought in the proposed error handling solutions is adaptability. Spoken language interfaces operate under a large variety of conditions: different performance in the underlying speech recognition and language understanding components, different and changing user populations, different qualities of the input lines, different costs for various types of errors, etc. We sought solutions that compensate for these differences by adapting to the characteristics of the domain in which the system operates. To this end, most of the error detection and recovery mechanisms proposed in this work have relied on supervised learning techniques.

Although these solutions are adapted to the training data, they also have at least two important drawbacks. First, they require a pre-existing a corpus of in-domain labeled data. Unfortunately, such corpora are difficult and expensive to collect and label, especially in the early stages of system development. Secondly, supervised learning techniques generally favor an off-line, or batch approach. A corpus is collected, manually labeled, and then model parameters are estimated from this data. The resulting model mirrors the properties of the training corpus, but does not respond well to changes in the system's environment and the underlying distribution of data. However, conversational spoken language interfaces are interactive systems that operate in dynamic environments. Consequently, shifts in the underlying distribution of the data are inevitable. To address these issues, in Chapter 5 we have proposed the use of a new learning paradigm, implicitly-supervised learning, in which the system obtains the desired supervision signal directly from naturally-ocurring patterns in the interaction. We have shown that this approach can be successfully applied to train confidence annotation models. We believe the approach is applicable in a number of other learning problems, not only in conversational spoken language interfaces, but also in the more general class of interactive systems. The paradigm eliminates the need for developer supervision and facilitates online adaptation and learning. We conjecture that it can supplement and even provide a strong alternative to existing learning approaches, and enable significant autonomous learning in interactive systems. The experiments we have conducted and results we have obtained on the confidence annotation task are very encouraging, but they represent only a first step towards understanding the properties, advantages and limitations of the proposed implicitly-supervised learning paradigm. In [16], we have articulated an extensive research program in this direction, centered on the following three questions: (1) how can systems make the most effective use of knowledge extracted from interaction patterns? (2) How can systems actively trigger these patterns to create learning opportunities but without having a significant negative impact on the interaction? (3) How can systems automatically identify novel knowledge-producing patterns in interaction and thereby increase the range of learning opportunities?

Lastly, it should be noted that the aim of the work described in this dissertation was to increase robustness in conversational spoken language interfaces by creating the mechanisms for seamlessly and efficiently recovering from errors. The high level question we addressed question: given that understanding-errors will occur, how can we detect them and set the conversation back on track? There is however a second important approach towards increased robustness that we have mentioned in the introduction chapter, but not addressed in this work: prevent the errors from happening altogether. These two approaches do not stand in opposition. Rather, a combined effort will lead to the best results. Error prevention is an area of research as important and challenging as error recovery. Based on lessons learned through this work, we expect that integrating high-level domain-specific information at the lower level of spoken dialog systems will play a key role in reducing the number of errors. Today, the "arrows" in the input processing pipeline in spoken language interfaces go mostly in one direction: speech recognition → language understanding → discourse interpretation. In order to reduce the number of errors, we will have to push more information (expectations) in the opposite direction. Simple solutions, like state-specific language models are already in use. The challenge lies in representing and integrating other types of discourse-level, domain-specific and world-knowledge into the search processes that govern the lower recognition and understanding levels. Accomplishing this integration in a general fashion remains an open area of research.

Appendix A

Sample conversations with RavenClaw/Olympus systems

This appendix contains several sample conversations with several RavenClaw-Olympus spoken dialog systems: RoomLine, Let's Go! Public, LARRI and Con-Quest.

§ Conversation with RoomLine

1 S: Welcome to RoomLine, the automated conference room reservation system within S C S. How may I help you today?
2 U: *I need a room for next Friday morning*
3 S: I need you to be more specific and tell me from what time until what time you need the room.
4 U: *Help!*
5 S: I am currently trying to gather enough information from you to make a conference room reservation. So far, I know you want a room for Friday, May seventh. I understood you want the room in the morning, but I need you to be more specific and tell me precisely from what time until what time you need the room. You can say something like 'from nine until eleven'.
6 U: *from ten until eleven thirty*
7 S: I found 4 rooms available Friday, May seventh between 10 and 11:30 a.m. Would you like a small or a large room?
8 U: *I want one which can seat thirty people*
9 S: 2 of these 4 rooms can accommodate 30 people: Newell Simon 1507, and Wean Hall 4623. Would you like a reservation for any of these rooms?
10 U: *sure*
11 S: Which room do you want?
12 U: *what were the rooms again?*
13 S: The list of available rooms we are discussing is : Newell Simon 1507, and Wean Hall 4623. These rooms are available Friday, May seventh between 10 and 11:30 a.m. Which room do you want?

14 U: *which one has a projector?*
15 S: Both these rooms have projection units. Which room do you want?
16 U: *I'll take any of them*
17 S: One choice would be Newell Simon 1507. This room can accommodate 50 people, and has a projector, a whiteboard and network access. Do you want a reservation for Newell Simon 1507?
18 U: *sure*
19 S: Okay, I will now make the reservation for Newell Simon 1507. Could I have your name please? If you are not registered with the system, just say 'guest'.
20 U: *guest*
21 S: Okay, I have successfully completed the reservation. Would you like me to summarize it for you?
22 U: *yes please*
23 S: I have reserved Newell Simon 1507 for Friday, May 7th, between 10 and 11:30 a.m. This room can accommodate 50 people, and has a projector, a whiteboard and network access. Is there anything else I could do for you today?
24 U: *no*
25 S: Thank you for calling. Have a nice day.

§ Conversation with Let's Go! Public

1 S: Welcome to the CMU Let's Go Bus Information System. To get help at any time, just say 'Help!' or press zero. What can I do for you?
2 U: *next 54C leaving from leaving from 18th and Arlington headed to the Junction*
3 S: the 54C … Did I get that right?
4 U: *yes*
5 S: Alright. Leaving from 18th and Arlington … Is this correct?
6 U: *yes*
7 S: Alright. Going to the Junction … Did I get that right?
8 U: *yes*
9 S: Right. When are you going to take that bus?
10 U: *now*
11 S: You want the next bus. Is that correct?
12 U: *yes*
11 S: Alright. Hold on … The next 54C leaves South 18th Street at Arlington at 7:51 p.m.

§ Conversation with LARRI

1 S: Hi, this is LARRI. State your name please
2 U: *Hi my name is John Doe*
3 S: Hello John Doe. Before you start on your scheduled work, we need you to finish an urgent task
4 U: *ok*

5 S: Let's reprioritize your schedule: first you will perform the INS Ground Initiated BIT Test on plane 301. You last performed this task 3 months ago. Is that correct?
6 U: *yes, that's right*
7 S: Then you will do the BXF radar check on plane 304. You last performed this task 2 weeks ago. Is that correct?
8 U: *affirmative*
9 S: When you are ready to start working on the INS Bit Test, please say ready to work.
10 U: *ready to work*
11 S: … [system presents task summary]
12 S: Step A: On RDDI press and release the MENU pushbutton switch until HSI pushbutton switch option appears.
13 U: *done*
14 S: Does the RDDI menu display the HSI?
15 U: *affirmative*
16 S: Step B: Now press the HSI pushbutton switch.
17 U: *Ok. RDDI displays the HSI*
18 S: Ok. Step C: Press the DATA pushbutton switch.
19 U: *…[long timeout] …*
20 S: Did you push the DATA pushbutton?
21 U: *yes, I just did*
22 S: Does the RDDI display DATA option display with the WYPT option boxed?
23 U: *yes*
24 S: Ok. Step D: Select WYPT 0.
25 U: *okay. WPT is 0 and the latitude is local*
26 S: Good. Skipping to step J. On RDDI press and release MENU pushbutton switch until BIT pushbutton switch option appears.
27 U: …

§ Conversation with ConQuest

1 S: Welcome to ConQuest […] I can give you information about papers, sessions, special events and announcements at InterSpeech 2006. You can also cast a vote for the best paper award. How can I help you?
2 U: *are there any papers on spoken dialog systems today*
3 S: There are 23 papers on spoken dialog systems today. These papers are being presented in 3 sessions: Dialog and Discourse, Language Modeling for Spoken Dialog Systems and Spoken Dialog Systems I. To browse this list of papers, please tell me the author name, the paper title, the time or the venue you're looking for. If you're not interested in any of these papers say 'quit'.
4 U: *which ones are in the afternoon*
5 S: There will be 11 papers on spoken dialog systems this afternoon. Would you like to hear information about all these papers?
6 U: *yes*
7 S: The list of papers on spoken dialog systems this afternoon is: Paper 1 - "Dynamic Extension of a Grammar-based Dialogue System: Constructing an All-Recipes Knowing Robot", Paper 2 - […] Would you like more information about any of these?

8 U: *tell me more about the third one*
9 S: Paper 3 is titled “System- versus user-initiative dialog strategy for driver information systems”, and was presented yesterday at 2 pm in the poster session in Allegheny Ballroom Foyer. This paper was written by Chantal Ackermann, and Marion Libossek. Would you like more information about any of these?
10 U: *no when is the Banquet*
11 S: The Banquet will be held Wednesday from 7pm to 9:30 pm in the Heintz History Center. The Heinz History Center is located on Smallman street between 12th and 13th streets. The History Center is a museum that relates and interprets 250 years of history in the Western Pennsylvania area. Is there anything else I can do for you?
13 U: *yes I’d like to cast a vote*
14 S: Please tell me the title of the paper you’d like to vote for ...
15 U: *I don’t really know the title but I think it had Diane as one of the main authors [non-understanding]*
16 S: Sorry, I didn’t catch that. Please dial the first six letters from the title of the paper you would like to vote for.
17 U: *I don’t know it, it’s a paper by Diane Litman*
18 S: There are 3 papers by Diane Litman. Paper 1 - “Discourse Structure and Speech Recognition Problems”, Paper 2 - […], Paper 3 - […] Would you like to vote for any of these?
19 U: *Discourse Structure and Speech Recognition Problems*
20 S: Okay, I’ve recorded your vote for the paper entitled “Discourse Structure and Speech Recognition Problems” What else can I do for you?
21 U: ...

Appendix B

Belief updating models

This appendix contains the five $BU_{n=1}^{m=1}$ models trained based on the RoomLine corpus, as discussed in subsection 6.4.5

REQ (request): full model		
	Coefficients	
Feature	r1/h1	other/h1
k	-0.79	3.57
barge_in	-2.08	-1.40
concept_id:date	11.29	9.80
concept_id:user_name	1.93	-13.91
dialog_state_id:RequestSpecificTimes	13.29	14.27
ih_diff_lexical	-1.55	1.18
i_h2_avail	-21.71	-2.72
total_num_parses	-1.07	-0.41
srh_r1_confidence	4.09	1.76
srh_r1_confusability	5.81	1.70
srh_r1_prior	0.67	0.98
srh_r1_prior>1	-1.00	-6.38

NOA (no action): full model		
	Coefficients	
Feature	r1/h1	other/h1
k	1.79	6.90
srh_r1_confusability	5.17	-0.26
ivs=value	-2.96	-1.79
ivs=empty_no_hist	-2.33	-3.43
word_num=1	-2.37	-0.04
word_num=2	-0.22	0.16
word_num=3	0.12	0.31
i_h1_prior	-0.58	-0.99
concept_id:date	0.77	6.43
i_h1_prior_gt_1	0.89	-3.45
srh_r1_explicitly_disconfirmed_already	-5.89	1.33
concept_id:_ChooseAnyRoom_trigger	16.31	3.37
i_h1_confusability	-4.52	-3.77
ih_diff_lexical	-1.15	-0.51
srh_r1_prior	0.28	-0.02
srh_h_h1_avail	-1.68	-4.05
lex:THAT'S	1.21	2.52
concept_id:size	0.80	8.15
dialog_state_id:HowMayIHelpYou	-0.25	-1.59
h_avg_confidence	3.25	1.29
i_h1_explicitly_confirmed_already	-13.68	-14.84
word_num	0.40	0.03
lm_score	0.00	0.00

EC (explicit confirm): full model		
	Coefficients	
Feature	r1/h1	other/h1
k	-15.96	3.61
answer_type:yes	-12.67	-5.91
answer_type:no	4.56	3.15
answer_type:other	1.21	-0.75
concept_id:equip	6.96	4.43
i_h1_confusability	-3.67	-4.81
ih_diff_lexical_one_word	-15.99	-1.17
lexw1:SMALL	17.64	20.27
srh_r1_avail	18.85	0.41

IC (implicit confirm): full model		
	Coefficients	
Feature	r1/h1	other/h1
k	-16.83	3.76
mark_confirm	0.32	-1.75
mark_disconfirm	3.40	1.58
i_h1_confidence	0.39	-3.63
i_h1_confusability	-4.18	-4.55
lex:THREE	-2.26	-2.69
srh_r1_avail	20.89	1.70
turn_number	0.00	0.03

UIC (unplanned implicit confirm): full model		
	Coefficients	
Feature	r1/h1	other/h1
k	-29.76	3.46
barge_in	0.72	0.78
concept_id:equip	15.76	1.66
i_h1_confusability	-2.12	-4.73
i_h1_explicitly_confirmed_already	-16.93	-19.12
i_h1_prior>1	-1.59	-2.50
mark_disconfirm	1.44	0.47
mark_confirm	-1.05	-0.77
srh_r1_avail	30.84	0.73
word_num>2	2.82	0.37

Bibliography

[1] Aist, G., Bohus, D., Boven, B., Campana, E., Early, S., and Phan, S., 2004 *Initial Development of a Voice-Activated Astronaut Assistant for Procedural Tasks: From Need to Concept to Prototype.* Journal of Interactive Instruction Development. **16**(3): p. 32-36.

[2] Aist, G., Dowding, J., Hockey, B. A., Rayner, M., Hieronymus, J., Bohus, D., Boven, B., Blaylock, N., Campana, E., Early, S., Gorrell, G., and Phan, S., 2003 - *Talking through procedures: An intelligent Space Station procedure assistant*. in Proceedings of EACL-2003. Budapest, Hungary.

[3] Aust, H. and Schroer, O., 1998 - *An overview of the Philips dialog system*. in Proceedings of DARPA Broadcast News Transcription and Understanding Workshop. Lans-downe, VA.

[4] Balakirsky, S., Scrapper, C., Carpin, S., and Lewis, M., 2006 - *UsarSim: providing a framework for multirobot performance evaluation*. in Proceedings of PerMIS.

[5] Balentine, B. and Morgan, D., *How to Build a Speech Recognition Application: A Style for Telephony Dialogues*. 1999: Enterprise Integration Group.

[6] Benjamini, Y. and Hochberg, Y., 1995 *Controlling the false discovery rate: a practical and powerful approach to multiple testing.* Journal of the Royal Statistics Society. **B**(57): p. 289-300.

[7] Bennett, P. 2006 - *Building Reliable Metaclassifiers for Text Learning*, in Computer Science, Carnegie Mellon University: Pittsburgh, PA.

[8] Black, A. and Lenzo, K., 2000 *Building Voices in the Festival Speech System.* http://festvox.org/bsv/

[9] Bohus, D. 2002 - *Integrating Multiple Knowledge Sources for Utterance-Level Confidence Annotation in the CMU Communicator Spoken Dialog System*, in Technical Report CS-190, Carnegie Mellon University: Pittsburgh, PA.

[10] Bohus, D., 2007 *On-line List of Spoken Dialog Systems.* http://www.cs.cmu.edu/~dbohus/SDS

[11] Bohus, D., Puerto, S. G., Huggins-Daines, D., Keri, V., Krishna, G., Kumar, R., Raux, A., and Tomko, S., 2007 - *Conquest – an Open-Source Dialog System for Conferences*. in Proceedings of HLT'07. Rochester, NY.

[12] Bohus, D., Raux, A., Harris, T., Eskenazi, M., and Rudnicky, A., 2007 - *Olympus: an open-source framework for conversational spoken language interface research*. in Proceedings of HLT-2007. Rochester, NY.

[13] Bohus, D. and Rudnicky, A., 2002 - *LARRI: A Language-Based Maintenance and Repair Assistant*. in Proceedings of IDS'02. Kloster Irsee, Germany.

[14] Bohus, D. and Rudnicky, A., 2003 - *RavenClaw: Dialog Management Using Hierarchical Task Decomposition and an Expectation Agenda*. in Proceedings of Interspeech-2006. Geneva, Switzerland.

[15] Bohus, D. and Rudnicky, A., 2005 - *Sorry, I Didn't Catch That! - an Investigation of Non-understanding Errors and Recovery Strategies*. in Proceedings of SIGdial-2005. Lisbon, Portugal.

[16] Bohus, D., Rudnicky, A., and Gordon, G. 2007 - *Implicit Learning in Spoken Language Interfaces*, Carnegie Mellon University.

[17] Bos, J., Klein, E., Lemon, O., and Oka, T., 2003 - *DIPPER: Description and Formalisation of an Information-State Update Dialogue System Architecture*. in Proceedings of SIGdial'03. Sapporo, Japan.

[18] Bosch, A. v. d., Krahmer, E., and Swerts, M., *Detecting problematic turns in human-machine interaction: rule-induction versus memory-based learning approaches.*

[19] Bourguet, M.-L., *Towards a Taxonomy of Error-Handling: Strategies in Recognition-Based Multimodal Human-Computer Interfaces.*

[20] Bousquet-Vernhettes, C., Privat, R., and Vigouroux, N., 2003 - *Error handling in spoken dialogue systems: toward corrective dialogue*. in Proceedings of Workshop on Error Handling in Spoken Dialogue Systems. Chateau d'Oex-Vaud, Switzerland.

[21] CALO, 2007 *CALO Project Website.* http://www.ai.sri.com/project/CALO

[22] Carpenter, P., Jin, C., Wilson, D., Zhang, R., Bohus, D., and Rudnicky, A., 2001 - *Is This Conversation on Track?* in Proceedings of Eurospeech-2001. Aalborg, Denmark.

[23] Cassell, J., Bickmore, T., Billinghurst, M., Campbell, L., Chang, K., Vilhjalmsson, H., and Yan, H., 1999 - *Embodiment in Conversational Interfaces: Rea*. in Proceedings of CHI'99. Pittsburgh, PA.

[24] Cepstral, L., 2005 *SwiftTM: Small Footprint Text-to-Speech Synthesizer.* http://www.cepstral.com.

[25] Choularton, S., 2005 - *Investigating the Acoustic Sources of Speech Recognition Errors*. in Proceedings of Interspeech'05. Lisbon, Portugal.

[26] Choularton, S. and Dale, R., 2004 - *User Responses to Speech Recognition Errors: Consistency in Behavior across Domains*. in Proceedings of SST'04.

[27] Clark, H. H. and Schaefer, E. F., 1989 *Contributing to Discourse.* Cognitive Science. **13**.

[28] Cohen, I. and Goldszmidt, M., 2004 - *Properties and benefits of calibrated classifiers*. in Proceedings of EMCL/PKDD. Pisa, Italy.

[29] Cole, R., 1999 - *Tools for Research and Education in Speech Science*. in Proceedings of International Conference of Phonetic Sciences. San Francisco, CA.

[30] Cox, S. and Dasmahapatra, S., 2000 - *A Semantically-Based Confidence Measure for Speech Recognition*. in Proceedings of ICSLP.

[31] Cuayahuitl, H. and Serridge, B., 2002 - *Out-of-Vocabulary Word Modeling and Rejection for Spanish Keywork Spotting Systems*. in Proceedings of 2nd Mexican International Conference on Artificial Intelligence.

[32] DeGroot, M. and Fienberg, S., 1983 *The comparison and evaluation of forecasters.* The Statistician(32): p. 12-22.

[33] Dowding, J., Gawron, J., Appelt, D., Cherny, L., Moore, R., and Moran, D., 1993 - *Gemini: A Natural Language System for Spoken Language Understanding*. in Proceedings of ACL'93. Columbus, OH.

[34] Ferguson, G. and Allen, J., 1998 - *TRIPS: An Intelligent Integrated Problem-Solving Assistant*. in Proceedings of AAAI'98. Madison, WI.

[35] Filisko, E. and Seneff, S., 2004 - *Error Detection and Recovery in Spoken Dialogue Systems*. in Proceedings of Workshop for Spoken Language Understanding for Conversational Systems. Boston, MA.

[36] Filisko, E. and Seneff, S., 2006 - *Learning Decision Models in Spoken Dialogue Systems via User Simulation*. in Proceedings of AAAI Workshop on Statistical and Empirical Approaches for Spoken Dialog Systems. Boston, Massachusetts.

[37] Freund, Y. and Shapire, R., 1996 - *Experiments with a new boosting algorithm*. in Proceedings of ICML'96.

[38] Gabsdil, M. and Lemon, O., 2004 - *Combining Acoustic and Pragmatic Features to Predict Recognition Performance in Spoken Dialogue Systems*. in Proceedings of ACL.

[39] Goldberg, J., Ostendorf, M., and Kirchhoff, K., 2003 - *The Impact of Response Wording in Error Correction Subdialogs*. in Proceedings of ISCA Workshop on Error Handling in Spoken Dialogue Systems. Chateaux d'Oex, Switzerland.

[40] Good, I. J., 1952 *Rational Decisions.* Journal of the Royal Statistics Society. **B**.

[41] Gorin, A. L., Riccardi, G., and Wright, J. H., 1997 *How May I Help You?* Speech Communication. **23**: p. 113-127.

[42] Gorrell, G., Lewin, I., and Rayner, M., 2002 - *Adding Intelligent Help to a Mixed Initiative Spoken Dialogue System*. in Proceedings of ICSLP'02. Denver, Colorado.

[43] Gruenstein, A., Wang, C., and Seneff, S., 2005 - *Context-sensitive statistical language modeling*. in Proceedings of Interspeech-2005. Lisbon.

[44] Guillevic, D., Gandrabur, S., and Normandin, Y., 2002 - *Robust Semantic Confidence Scoring*. in Proceedings of ICSLP.

[45] Gustafson, J., Bell, L., Beskow, J., Boye, J., Carlson, R., Edlund, J., Granstorm, B., House, D., and Wiren, M., 2000 - *AdApt – a multimodal conversational dialogue system in an apartment domain*. in Proceedings of ICSLP'00. Beijing, China.

[46] Harris, R. A., *Voice Interaction Design: crafting the new conversational speech systems*. 2004: Morgan Kaufmann Publishers. 624.

[47] Harris, T., Banerjee, S., and Rudnicky, A., 2005 - *Heterogeneous Multi-Robot Dialogues for Search Tasks*. in Proceedings of AAAI Spring Symposium: Dialogical Robots. Palo Alto, CA.

[48] Hartikainen, M., Salonen, E.-P., and Turunen, M., 2004 - *Subjective Evaluation of Spoken Dialogue Systems Using SER VQUAL Method*. in Proceedings of ICSLP'04. Jeju Island, Korea.

[49] Hazen, T., Seneff, S., and Polifroni, J., 2002 *Recognition confidence scoring and its use in speech understanding systems.* Computer Speech and Language.

[50] Heer, J., Good, N., Ramirez, A., Davis, M., and Mankoff, J., 2004 - *Presiding Over Accidents: System Direction and Human Action*. in Proceedings of CHI'04. Vienna, Austria.

[51] Higashinaka, R., Nakano, M., and Aikawa, K., 2003 - *Corpus-Based Discourse Understanding in Spoken Dialogue Systems*. in Proceedings of ACL-2003.

[52] Higashinaka, R., Sudoh, K., and Nakano, M., 2005 - *Incorporating Discourse Features into Confidence Scoring of Intention Recognition Results in Spoken Dialogue Systems*. in Proceedings of ICASSP.

[53] Hirchberg, J., Litman, D., and Swerts, M., 2004 *Prosodic and other cues to speech recognition failures.* Speech Communication.

[54] Hirschberg, J., Litman, D., and Swerts, M., 2001 - *Identifying User Corrections Automatically in Spoken Dialogue Systems*. in Proceedings of NAACL-2001. Pittsburgh, PA.

[55] Hone, K. S. and Graham, R., 2000 *Towards a tool for the subjective assessment of speech system interfaces (SASSI).* Natural Language Engineering. **6**(3/4): p. 287-305.

[56] Horvitz, E. and Paek, T., 1999 - *A Computational Architecture for Conversation*. in Proceedings of Seventh International Conference on User Modeling. Banff, Canada.

[57] Horvitz, E. and Paek, T., 2000 - *DeepListener: Harnessing Expected Utility to Guide Clarification Dialog in Spoken Language Systems*. in Proceedings of ICSLP-2000. Beijing, China.

[58] Huang, X., Alleva, F., Hon, H.-W., Hwang, M.-Y., Lee, K.-F., and Rosenfeld, R., 1992 *The SPHINX-II Speech Recognition System: an overview.* Computer Speech and Language. **7**(2): p. 137-148.

[59] Jung, H., Allen, J., Chambers, N., Galescu, L., Swift, M., and Taysom, W., 2006 - *One-Shot Procedure Learning from Instruction and Observation*. in Proceedings of International FLAIRS Conference: Special Track on Natural Language and Knowledge Representation.

[60] Kaelbling, L., *Learning in Embedded Systems*. 1993, Boston, MA: MIT Press.

[61] Kawahara, T. and Komatani, K., 2000 - *Flexible mixed-initiative dialogue management using concept-level confidence measures of speech recognizer output*. in Proceedings of COLING'00. Saarbrucken, Germany.

[62] Krahmer, E., Swerts, M., Theune, M., and Weegels, M., *Error Detection in Human-Machine Interaction*, in *Speaking. From Intention to Articulation*. 1999, MIT Press.

[63] Krahmer, E., Swerts, M., Theune, M., and Weegels, M., 2001 *Error Detection in Spoken Human-Machine Interaction.* International Journal of Speech Technology. **4**(1): p. 19-30.

[64] Larsson, S. 2002 - *Issue-based dialog management*, Goteborg University.

[65] Lemon, O., Cavedon, L., and Kelly, B., 2003 - *Managing Dialogue Interaction: A Multi-Layered Approach*. in Proceedings of SigDIAL'03. Sapporo, Japan.

[66] Lemon, O., Gruenstein, A., Cavedon, L., and Peters, S., 2002 - *Multi-Tasking and Collaborative Activities in Dialogue Systems*. in Proceedings of SigDIAL'02. Philadelphia, PA.

[67] Levin, E., Pieraccini, R., and Eckert, W., 2000 *A Stochastic Model of Human-Machine Interaction for Learning Dialogue Strategies.* IEEE Transactions on Speech and Audio Processing. **8**(1).

[68] Levow, G.-A., 1998 - *Characterizing and Recognizing Spoken Corrections in Human-Computer Dialogue*. in Proceedings of COLING-ACL.

[69] Litman, D., Hirschberg, J., and Swerts, M., 2001 - *Predicting User Reactions to System Error*. in Proceedings of ACL-2001.

[70] Litman, D., Walker, M., and Kearns, M., 1999 - *Automatic Detection of Poor Speech Recognition at the Dialogue Level*. in Proceedings of.

[71] Mankoff, J., Hudson, S., and Abowd, G., 2000 - *Providing Integrated Toolkit-Level Support for Ambiguity in Recognition-Based Interfaces*. in Proceedings of CHI-2000. The Hague, Amsterdam.

[72] Martin, D., Cheyer, A., and Moran, D., 1999 *The Open Agent Architecture: A Framework for Building Distributed Software Systems.* Applied Artificial Intelligence. **13**(1/2): p. 91-128.

[73] McRoy, S. and Hirst, G., 1995 *The Repair of Speech Act Misunderstandings by Abductive Inference.* Computational Linguistics. **21**(4): p. 435-478.

[74] Mitchell, T., *Machine Learning*. 1997: McGraw Hill.

[75] Moreno, P., Logan, B., and Raj, B. 2001 - *A Boosting Approach for Confidence Scoring*, in MERL Tech Report, Mitsubishi Electric Research Laboratories.

[76] Myers, R. H., Montgomery, D. C., and Vining, G., *Generalized Linear Models: With Applications in Engineering and the Sciences*. Wiley series in probability and statistics, ed. Wiley-Interscience. 2001. 424.

[77] Nuance, 2007 *Nuance Speech Recognition.* http://www.nuance.com

[78] Nuance, 2007 *Nuance SpeechObjects.* http://www.w3.org/TR/speechobjects/

[79] Oviatt, S., 2000 *Taming recognition errors with a multimodal interface.* Communications of the ACM. **43**(9): p. 45-51.

[80] Paek, T., 2001 - *Empirical methods for evaluating dialog systems*. in Proceedings of SIGdial. Aalborg, Denmark.

[81] Paek, T., 2003 *Toward a Taxonomy of Communication Errors.*

[82] Paek, T., 2006 - *Reinforcement learning for spoken dialogue systems: Comparing strengths and weaknesses for practical deployment*. in Proceedings of Interspeech-06 Workshop on "Dialogue on Dialogues - Multidisciplinary Evaluation of Advanced Speech-based Interactive Systems". Pittsburgh, PA.

[83] Paek, T. and Horvitz, E., 2000 - *Conversation as Action Under Uncertainty*. in Proceedings of Sixteenth Conference on Uncertainty and Artificial Intelligence. Stanford, CA.

[84] Pieraccini, R. and Levin, E., 1993 - *A learning approach to natural language understanding*. in Proceedings of NATO ASI Summer School. Bubion, Spain.

[85] Pradhan, S. and Ward, W., 2002 - *Estimating Semantic Confidence for Spoken Dialogue Systems*. in Proceedings of ICASSP.

[86] Raux, A. 2006 - *Flexible Turn-Taking for Spoken Dialogue Systems*, in Ph.D. Thesis Proposal, Carnegie Mellon University.

[87] Raux, A., Bohus, D., Langner, B., Black, A. W., and Eskenazi, M., 2006 - *Doing Research on a Deployed Spoken Dialogue System: One Year of Let's Go! Experience*. in Proceedings of InterSpeech'06. Pittsburgh, PA.

[88] Raux, A. and Eskenazi, M., 2004 - *Non-native Users in the Let's Go! Spoken Dialogue System: Dealing with Linguistic Mismatch*. in Proceedings of HLT-NAACL'04. Boston, MA.

[89] Raux, A., Langner, B., Bohus, D., Black, A., and Eskenazi, M., 2005 - *Let's Go Public! Taking a Spoken Dialog System to the Real World*. in Proceedings of Interspeech-2005. Lisbon, Portugal.

[90] RavenClaw-Olympus, 2007 *RavenClaw-Olympus web page.* http://www.ravenclaw-olympus.org

[91] Raymond, C., Bechet, F., De Mori, R., and Damnati, G., 2004 - *On the use of confidence for statistical decision in dialogue strategies*. in Proceedings of SigDIAL'04. Boston, MA.

[92] Rayner, M., Dowding, J., and Hockey, B. A., 2001 - *A baseline method for compiling typed unification grammars into context-free language models*. in Proceedings of Eurospeech'01. Aalborg, Denmark.

[93] Rayner, M., Hockey, B. A., Renders, J. M., Chatzichrisafis, N., and Farrell, K., 2005 - *A Voice Enabled Procedure Browser for the International Space Station*. in Proceedings of 44th Annual

Meeting of the Association for Computational Linguistics (interactive poster and demo track). Ann Arbor, MI.

[94] Rich, C. and Sidner, C., 1998 *COLLAGEN: A Collaboration Manager for Software Interface Agents.* An International Journal: User Modeling and User-Adapted Interaction. **8**(3/4): p. 315-350.

[95] Rich, C., Sidner, C., and Lesh, N. 2001 - *COLLAGEN: Applying Collaborative Discourse Theory to Human-Computer Interaction*, in AI Magazine. p. 15-25.

[96] Rickel, J., Lesh, N., Rich, C., Sidner, C., and Gertner, A., 2001 - *Building a Bridge between Intelligent Tutoring and Collaborative Dialogue Systems*. in Proceedings of Tenth International Conference on AI in Education.

[97] Rieser, V., Korbayová, I.-K., and Lemon, O., 2005 - *A Corpus Collection and Annotation Framework for Learning Multimodal Clarification Strategies*. in Proceedings of SIGdial'05. Lisbon, Portugal.

[98] Rieser, V. and Lemon, O., 2006 - *Using Machine Learning to Explore Human Multimodal Clarification Strategies*. in Proceedings of COLING/ACL-06. Sydney, Australia.

[99] Rosenfeld, R., Olsen, D., and Rudnicky, A. 2000 - *A Universal Human-Machine Speech Interface*, in Technical Report, Carnegie Mellon University: Pittsburgh, PA.

[100] Rotaru, M. and Litman, D., 2005 - *Interactions between Speech Recognition Problems and User Emotions*. in Proceedings of Interspeech-2005. Lisbon, Portugal.

[101] Rudnicky, A., Thayer, E., Constantinides, P., Tchou, C., Stern, R., Lenzo, K., Xu, W., and Oh, A., 1999 - *Creating natural dialogs in the Carnegie Mellon Communicator system*. in Proceedings of Eurospeech'99.

[102] San-Segundo, R., Pellom, B., Hacioglu, K., and Ward, W., 2001 - *Confidence Measures for Spoken Dialogue Systems*. in Proceedings of ICASSP.

[103] San-Segundo, R., Pellom, B., and Ward, W., 2002 - *Confidence Measure for Dialogue Management in the CU Communicator System*. in Proceedings of ICASSP 2000.

[104] Sarikaya, R., Gao, Y., and Erdogan, H., 2005 *Semantic Confidence Measurement for Spoken Dialog Systems.* IEEE Transactions on Speech and Audio Processing.

[105] Savin, N. E., 1980 *The Bonferroni and the Scheffe Multiple Comparison Procedures.* Review of Economic Studies. **47**(1): p. 255-273.

[106] Schatzmann, J., Georgila, K., and Young, S., 2005 - *Quantitative Evaluation of User Simulation Techniques for Spoken Dialogue Systems*. in Proceedings of SIGdial'05. Lisbon, Portugal.

[107] Scheffler, K. 2002 - *Automatic Design of Spoken Dialogue Systems*, in Cambridge University, Cambridge University: Cambridge, UK.

[108] Scheffler, K. and Young, S., 2002 - *Automatic learning of dialogue strategy using dialogue simulation and reinforcement learning*. in Proceedings of HLT-2002. San Diego, California.

[109] Schlangen, D., 2004 - *Causes and Strategies for Requesting Clarification in Dialogue*. in Proceedings of SIGdial. Boston, MA.

[110] Seneff, S., Chuu, C., and Cyphers, D. S., 2000 - *Orion: From On-line Interaction to Off-line Delegation*. in Proceedings of ICSLP'00. Beijing, China.

[111] Seneff, S., Hurley, E., Lau, R., Pao, C., Schmid, P., and Zue, V., 1998 - *Galaxy-II: A reference architecture for conversational system development*. in Proceedings of ICSLP'98. Sydney, Australia.

[112] Shi, Y. and Zhou, L., 2005 - *Error Detection Using Linguistic Features*. in Proceedings of HLT.

[113] Shin, J. and Narayanan, S., 2002 - *Analysis of User Behavior under Error Conditions in Spoken Dialogs*. in Proceedings of ICSLP-2002. Denver, CO.

[114] Singh, S., Litman, D., Kearns, M., and Walker, M., 2000 *Optimizing Dialogue Management with Reinforcement Learning: Experiments with the NJFun System.* Journal of Artificial Intelligence Research. **16**: p. 105-133.

[115] Skantze, G., 2003 - *Exploring Human Error Handling Strategies: Implications for Spoken Dialogue Systems*. in Proceedings of ISCA Workshop on Error Handling in Spoken Dialogue Systems. Chateau d□Oex Vaud, Switzerland.

[116] Skantze, G. and Edlund, J., 2004 - *Early error detection on word level*. in Proceedings of.

[117] Steidl, S., Hacker, C., Ruff, C., Batliner, A., Noth, E., and Haas, J., 2004 - *Looking at the Last Two Turns, I'd Say This Dialogue is Doomed - Measuring Dialogue Success*. in Proceedings of TSD.

[118] Stemmer, G., Steidl, S., Noth, E., Niemann, H., and Batliner, A., 2002 - *Comparison and Combination of Confidence Measures*. in Proceedings of TSD.

[119] Sturm, J. and Boves, L., 2005 *Effective error recovery strategies for multimodal form-filling applications.* Speech Communication(45): p. 289-303.

[120] Sundbald, O. and Sundbald, Y., 1998 - *Olga: a Multimodal Interactive Information Assistant*. in Proceedings of CHI'98.

[121] Swerts, M., Litman, D., and Hirschberg, J., 2000 - *Corrections in Spoken Dialogue Systems*. in Proceedings of ICSLP-2000. Beijing, China.

[122] Tomko, S. 2003 - *Speech Graffiti: Assessing the User Experience*, in Language Technologies Institute, Carnegie Mellon University.

[123] Tomko, S. 2007 - *Improving Interaction with Spoken Dialog Systems via Shaping*, in Language Technologies Institute, Carnegie Mellon University: Pittsburgh, PA.

[124] TrindiKit, 2007 *TrindiKit.* http://www.ling.gu.se/projekt/trindi/trindikit/

[125] Understanding, C. f. S. L., 2007 *CSLU Toolkit.* http://cslu.cse.ogi.edu/toolkit/

[126] VoiceXML, 2007 *VoiceXML specification.* http://www.voicexml.org

[127] Walker, M., Litman, D., Kamm, C., and Abella, A., 1997 - *PARADISE: A General Framework for Evaluating Spoken Dialogue Agents*. in Proceedings of ACL/EACL.

[128] Walker, M., Litman, D., Kamm, C., and Abella, A., 1998 *Evaluating Spoken Dialogue Systems with PARADISE: Two Case Studies.* Computer Speech and Language. **12**(3).

[129] Walker, M., Passonneau, R., and Boland, J., 2001 - *Quantitative and Qualitative Evaluation of the DARPA Communicator Spoken Dialogue Systems*. in Proceedings of ACL'01.

[130] Walker, M., Wright, J., and Langkilde, I., 2000 - *Using Natural Language Processing and Discourse Features to Identify Understanding Errors in a Spoken Dialogue System*. in Proceedings of ICML'00.

[131] Ward, W. and Issar, S., 1994 - *Recent improvements in the CMU spoken language understanding system*. in Proceedings of ARPA Human Language Technology Workshop. Plainsboro, NJ.

[132] Weegels, M., *User's Conceptions of Voice-Operated Information Services.* International Journal of Speech Technology. **3**(2): p. 75-82.

[133] Wessel, F., Schlutter, R., Macherey, K., and Ney, H., 2001 *Confidence Measures for Large Vocabulary Continuous Speech Recognition.* IEEE Transactions on Speech and Audio Processing. **9**(3).

[134] Wikipedia, 2007 *Gini coefficient.* http://en.wikipedia.org/wiki/Gini_coefficient

[135] Wikipedia, 2007 *Spearman's rank correlation coefficient.* http://en.wikipedia.org/wiki/Spearman's_rank_correlation_coefficient

[136] Williams, J. and Young, S., 2005 - *Scaling up POMDPs for Dialogue Management: the Summary POMDP Method*. in Proceedings of ASRU'05. San Juan, Puerto Rico.

[137] Xu, W. and Rudnicky, A., 2000 - *Language Modeling for Dialogue Systems*. in Proceedings of ICSLP'00. Beijing, China.

[138] Yankelovich, N., Levow, G.-A., and Marx, M., 1995 - *Designing SpeechActs: Issues in Speech User Interfaces*. in Proceedings of CHI'95. Denver, CO.

[139] Zhang, J. and Yang, Y., 2004 - *Probabilistic Score Estimation with Piecewise Logistic Regression*. in Proceedings of ICML-2004. Banff, Alberta, Canada.

[140] Zhang, R. and Rudnicky, A., 2006 - *A New Data Selection Approach for Semi-Supervised Acoustic Modeling*. in Proceedings of ICASSP'06. Toulouse, France.

[141] Zollo, T., 1999 - *A Study of Human Dialogue Strategies in the Presence of Speech Recognition Errors*. in Proceedings of AAAI Fall Symposium on Psychological Models of Communication in Collaborative Systems.

[142] Zue, V., Seneff, S., Glass, J., Polifroni, J., Pao, C., Hazen, T., and Hetherington, L., 2000 *JUPITER: A Telephone-Based Conversational Interface for Weather Information.* IEEE Transactions on Speech and Audio Processing. **8**(1).