Toolkit to Support Intelligibility in Context-Aware Applications

Brian Y. Lim, Anind K. Dey
Human-Computer Interaction Institute
Carnegie Mellon University
5000 Forbes Ave., Pittsburgh, PA 15213
{byl, anind}@cs.cmu.edu

ABSTRACT
Context-aware applications should be intelligible so users can better understand how they work and improve their trust in them. However, providing intelligibility is non-trivial and requires the developer to understand how to generate explanations from application decision models. Furthermore, users need different types of explanations and this complicates the implementation of intelligibility. We have developed the Intelligibility Toolkit that makes it easy for application developers to obtain eight types of explanations from the most popular decision models of context-aware applications. We describe its extensible architecture, and the explanation generation algorithms we developed. We validate the usefulness of the toolkit with three canonical applications that use the toolkit to generate explanations for end-users.

Author Keywords
Context-awareness, intelligibility, explanations, toolkits.

ACM Classification Keywords
H5.m. Information interfaces and presentation (e.g., HCI): Miscellaneous.

General Terms
Algorithms, Human Factors.

INTRODUCTION
Context-aware applications [11] make use of sensed inputs and contexts, coupled with intelligent decision-making, in order to automatically and calmly [43] adapt to serve users better. However, the implicit nature of context sensing, and the growing complexity of models (*e.g.*, rules, hidden Markov models) underlying context-aware applications make it difficult for users to understand them (*e.g.*, [42]). This can frustrate users [4], and cause them to lose trust in the applications [29]. To counter this, it is important to make context-aware applications *intelligible* [5, 25, 26] by automatically providing explanations of application behavior. In fact, this has been done in applications from other domains (*e.g.*, recommender systems [20], end-user debugging [22], user interfaces [30], user modeling [10]), and been found to improve user trust and acceptance of these applications.

Several reviews have shown that users desire a wide range of explanations (*e.g.*, [17, 23, 26, 27]) and specifically, Lim & Dey [25] describe a set of ten explanation types that context-aware applications should provide for end-users. The already challenging task of generating some explanations from applications is made more challenging with this requirement for many explanation types. In this work, we designed and implemented the Intelligibility Toolkit that automatically generates explanations from models at the infrastructure level, so application developers do not have to derive explanations themselves.

Our contributions are:

(1) An architecture for generating a wide range of explanations drawn from Lim & Dey [25], including Why, Why Not, How To, What, What If, Inputs, Outputs and Certainty. Our current implementation extends a popular toolkit for building context-aware applications [11, 13] and supports the four most popular model types (rules, decision trees, naïve Bayes, and hidden Markov models).

(2) A library of reference implementations of explanation generation algorithms we developed to extract any of the 8 explanation types from any of the four models we support.

(3) Automated support for the recommendations from [25] that promotes good design practice by making the most contextually appropriate explanations easy for developers to acquire. Applications can automatically obtain the most appropriate explanations given the contextual situation.

As we will show, these contributions satisfy the requirements laid out by Olsen for adding value to user interface architectures [32]: (i) importance, (ii) problem not previously solved, (iii) has *generality* across a range of explanation types and decision model types, (iv) *reduces solution viscosity* through increased *flexibility* for rapid prototyping of explanations, (v) *empowers new design participants* by making it easier to provide explanations, and (vi) demonstrates *power in combination* by supporting combinations of explanation types as building blocks.

UbiComp '10, Sep 26–Sep 29, 2010, Copenhagen, Denmark.

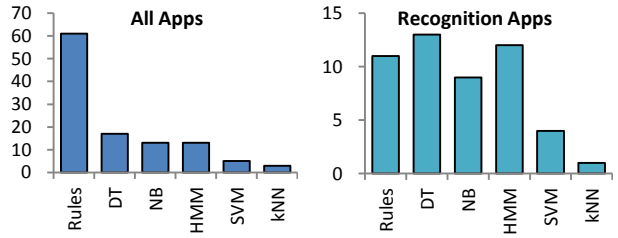

Figure 1: (Left) Counts of model types used in 109 of 114 reviewed context-aware applications. (Right) Counts for 50 recognition applications; classifiers are used most often for applications that do recognition. Key: decision tree (DT), naïve Bayes (NB), hidden Markov models (HMM), support vector machines (SVM), k-Nearest Neighbor (kNN).

The paper is organized as follows: we review context-aware applications published in recent years from a number of premier conferences to ascertain the most popular decision models. Then we present a discussion of the explanation types we seek to provide. Next, we describe the Intelligibility Toolkit and how it provides explanations for context-aware applications. We detail the explanation generation algorithms we developed for the four most popular decision models. We *validate* the toolkit through demonstration applications and the application of Olsen's infrastructure guidelines. We then discuss other toolkit features that would be valuable to support, and compare the toolkit with related work. We end with discussing opportunities for new research given the toolkit.

REVIEW OF DECISION MODELS USED IN CONTEXT-AWARE COMPUTING

We sought to learn what the most popular context-aware decision models are so that we could support these in the Intelligibility Toolkit. We reviewed literature from three major conferences over at least five years: CHI 2003-2009, Ubicomp 2004-2009, and Pervasive 2004-2009. We found that there were four popular models among the 114 context-aware applications reviewed: rules, decision trees, naïve Bayes, and hidden Markov models (see Figure 1). We found some use of Support Vector Machines, and clustering techniques (*e.g*, k-Nearest Neighbor), but not substantial.

Rules: We classify applications as rule-based if the authors state that their applications were based on rules (*e.g.*, if/else logic), or that they were based on simple mapping associations of IDs to entities (*e.g.*, RFID). Developer specified rules are the most popular decision models used in context-aware applications. They are popular in the following domains: activity recognition (*e.g.*, [41]), adaptation / personalization (*e.g.*, [40]), awareness / monitoring (*e.g.*, [12]), reminders (*e.g.*, [6]), location guides (*e.g.*, [31]), and persuasion (*e.g.*, [16]). Also, the number of rule-based toolkits for context-aware applications indicates the popularity of rules (*e.g.*, [1, 3, 13, 18]).

Decision tree classifiers (*e.g.*, [35]) learn a tree from a dataset. A decision tree infers an output by deciding on a specific input feature at each node as it traverses down and returns a decision once it reaches a leaf. Decision trees are popular for their simplicity of use, interpretability, and good runtime performance. Decision trees are popular in applications to recognize: identity / ability (*e.g.*, [8]), interruptibility (*e.g.*, [2, 42]), mobility (*e.g.*, [45]), *etc.*

Naïve Bayes is a probabilistic classifier that applies Bayes theorem to model the probability of the output of a system given the inputs. It applies a naïve assumption that features are conditionally independent of one another. Training and runtime performance are fast. Naïve Bayes classifiers have been used to recognize: physical activity (*e.g.*, [9]), domestic activity (*e.g.*, [39]), interruptibility (*e.g.*, [42]).

Hidden Markov models (HMM) [36] are Bayesian probabilistic classifiers that model the probability of a *sequence* of hidden states given a sequence of observations (input features with respect to time). First-order Markov models assume that only the previous state affects the next, and only the current state influences the current observation. HMMs have been used to model: physical (*e.g.*, [9]) and domestic activity (*e.g.*, [21]), gaze (*e.g.*, [7]).

Several applications also combine rules for higher level logic with classifiers for lower level recognition (*e.g.*, [16]).

INTELLIGIBILITY AND EXPLANATION TYPES

As we focus on these four decision models, we need to generate the various explanations that users want to receive. Lim *et al.* [25, 26] enumerated ten explanation types that are important to end-users of context-aware applications, and in this work we provide mechanisms to *automatically* extract eight of them from the applications. We review the different explanation types and their definitions.

Model-independent explanation types

If we represent a context-aware application as sensing inputs, maintaining system state, and producing an output, we can identify a set of explanations that are independent of the decision model and how it makes its decisions.

Inputs explanations inform users what input sensors (*e.g.*, thermostat, GPS coordinates) and information sources (*e.g.*, weather forecast, restaurant reviews website) that the application employs so that users can understand its scope. Inputs should be described by their name and possibly also with some description of what they mean or refer to.

Outputs explanations inform users what output options the application can produce. This lets users know what it can do or what states it can be in (*e.g.*, activity recognized as one of three options: sitting, standing, walking). This helps users understand the extent of the application's capabilities. Outputs explanations can also be used to help ask model-based explanations Why Not and How To (see below) by allowing the user to select an alternative desired output.

What explanations inform users of the current (or previous) system state in terms of output value; this makes the application state explicit. Input values are obtained by recursively asking What on the Inputs. When a user asks a why question, she may actually be asking for What.

What If explanations allow users to speculate what the application would do given a set of user-set input values.

Model-dependent explanation types

Explanations regarding the mechanism of the decision making process in the application are model-dependent, and would vary depending on the model used.

Why explanations inform users why the application derived its output value from the current (or previous) input values. For rule-based systems, this returns the conditions (rules) that were true such that the output was selected.

Why Not explanations inform users why an alternative output value was *not* produced given the current input values. They could provide users with enough information to achieve the alternative output value, but not necessarily so (*e.g.*, see the naïve Bayes explanation algorithm later).

How To explanations answer the question "In *general*, how can the application produce alternative output value X?" This is in contrast to asking when a specific event occurs.

Certainty explanations inform users how (un)certain the application is of the output value produced. They help the user determine how much to trust the output value.

We omit explaining about the Situation because it is domain specific and independent of application behavior. We omit explaining Control because this is already covered by Enactor Parameters [13].

INTELLIGIBILITY TOOLKIT

We have presented the four most popular context-aware decision model types, and laid out eight explanation types that are important for context-aware applications to provide. Here we describe the Intelligibility Toolkit that supports the automatic generation of these explanation types from these decision models. Context-aware applications built with the toolkit freely receive the capability of providing these generated explanations to end-users. The toolkit is designed to satisfy *requirements* inspired from Olsen's writings about infrastructure evaluation [32]:

R1) Lower barrier to providing explanations. With the toolkit, application developers do not need to know how to generate explanations from the most popular models used in context-aware applications. Explanation generation algorithms and heuristics are encapsulated into the toolkit.

R2) Flexibility of using explanations. Given the simplicity of invoking various explanation types, all types can be generated with the same level of ease. Developers can then concentrate on choosing the most suitable explanation for their applications and users. This supports rapid prototyping of providing explanation solutions to see which works best.

R3) Facilitate appropriate explanations automatically. Even with this rapid prototyping support, we encourage the use of appropriate explanations, particularly following the recommendations from Lim & Dey [25] of how different explanation types are more appropriate in various contexts.

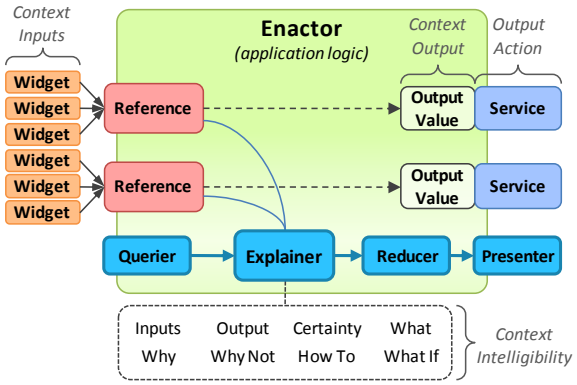

Figure 2: Architecture for handling rules and classifiers. The Intelligibility Toolkit adds four components to the Enactor framework of the Context Toolkit. Users ask for explanations with Querier, and invoke Explainer to generate explanations. The explanations may be simplified with a Reducer, and rendered through a Presenter.

R4) Support combining of explanations. Some explanation types depend on other types to give a complete explanation to the user. For example, a Why Not explanation needs to inform the user of the set of Output values so that she would ask only about what is possible in the application. Other than nesting explanations, explanations can also be combined to enhance user experience. For example, combining the How To and What If explanations can expedite users in finding good examples to learn how an application works (see Figure 7, right).

R5) Generalizability across (a) explanation types; (b) decision model types; and, (c) explanation provision. Although the toolkit currently covers a range of eight explanation types, four popular decision models and methods for simplifying and presenting explanations, like any toolkit, it is not comprehensive. The toolkit can be extended to support (a) new explanation types and new methods to explain the supported models (*e.g.*, competing methods to explain naïve Bayes are presented in [28, 34, 37]); (b) new model types (*e.g.*, SVM, clustering) as long as explanation generation algorithms can be developed for them; and (c) new ways to present explanations.

Architecture

The Intelligibility Toolkit (Figure 2), implemented in Java, leverages mechanisms in the Enactor framework [13] of the Context Toolkit [11]. Enactors contain the application logic of the context-aware application and contain References that monitor the state of input Widgets. Each Reference contains a *rule* and is triggered when its rule is satisfied. This is managed automatically by the discovery mechanism of the Context Toolkit.

We added an *output* property and a *list* of its values for the Enactor to represent its output value, and output options. Each output value is associated with a Reference.

Extension to support classifiers

As it was previously purely rule-based, we extended the Enactor framework to also support machine learning

classifiers. In a classifier-based set up, multiple References can be associated with a classifier, but each Reference is associated with a different output value. Each Reference is triggered when the classifier classifies the Widget state as the Reference's associated output value. We used Weka [19] for the decision tree and naïve Bayes classifiers, and Jahmm [15] for HMM classifiers.

Modifications to extend Intelligibility

Next, we describe four components we added to the Enactor framework to support a wider range of intelligibility (see architecture diagram in Figure 2).

Explainer

This is the main component of the Intelligibility Toolkit that contains the mechanisms and algorithms to generate explanations based on the decision model. There is a generic `Explainer` that generates explanations for model-independent types, and subclasses of Explainer for each of the four decision models supported (see next section on Explanation Generation Algorithms). While we have implemented one Explainer per model, additional Explainers can be developed to support other types of models we have not covered, and also to generate different explanation *methods* for existing model types. For example, we implemented the weights of evidence method from [34] to explain Bayesian models (naïve Bayes, HMM), but there are other explanation methods that could also provide different explanations, *e.g.*, [28, 37]. The use of Explainers and their standardized programming interfaces supports requirements R1 and R2. The creation of new Explainers for models or explanation types to plug into the toolkit supports R5a and R5b, respectively.

Explanation Struct (Expression)

We define an explanation in terms of one or multiple reasons (*e.g.*, multiple reasons for Why Not). Each reason can be a singular conditional (*e.g.*, one certainty value for a Certainty explanation) or a conjunction (*e.g.*, multiple conditionals for a Why explanation). The conditional is the atomic unit of an explanation (*e.g.*, certainty=90%, temperature<24°C). Furthermore, there can be negated conditionals (*e.g.*, not temperature≥24°C). Formally, we define explanations in Disjunctive Normal Form (DNF), *i.e.* a disjunction (OR) of conjunctions (ANDs) of conditionals (see Figure 3; see example in Figure 4). The standardization of explanation information supports R4 such that there is a standard way to pipe and feed different explanation types.

BestExplanationAdapter

We support requirement R3 by providing the `BestExplanationAdapter` component. It takes in the current context of the application (*e.g.*, appropriateness, and criticality) and returns the most suitable explanation types (*e.g.*, providing Why Not explanation of the second likeliest inference when the system accuracy is below a threshold).

Querier

While some explanation types (Inputs, Outputs) are constant once the application and model are defined and do not depend on events or instances, other types depend on what happened and what the user is asking. The base `Querier` takes the current inputs and output values and is used for explanation types What, Why, and Certainty. `AltQuerier` extends Querier and includes an alternative target output value to facilitate explanation types Why Not, and How To. `UserInputsQuerier` extends Querier and allows the setting of input values, supporting What If explanations. Queriers can be extended to employ different constraining mechanisms, such as querying based on time. Formally, they allow explanations to be constrained (*e.g.*, to convert How To to Why) allowing for different explanation types to be supported (requirement R5a).

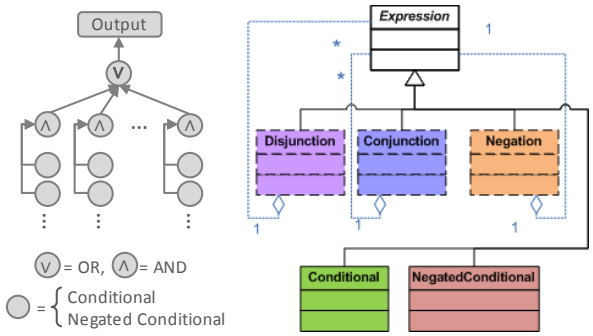

Figure 3: Schematic of explanation in Disjunctive Normal Form (DNF), and UML diagram of Explanation Struct format to programmatically represent explanations.

Reducer

Explanations generated from Explainers may be unwieldy to an end-user in two ways: (i) too many reasons (*e.g.*, numerous ways to achieve a target output value), and (ii) each reason being too long (*e.g.*, numerous inputs with required values to cause the output value). The latter case occurs when many sensors and feature values are used to build the models, as is the case for accurate learned systems. Reducer components simplify the Explanation Struct so that the explanation is more interpretable to users. To reduce the number of reasons, we implemented two types of `DisjunctionReducers`: `FirstDReducer` that just takes the first conjunction in the order of disjunctions of the explanation, and `ShortestDReducer` selects the shortest conjunction reason. To reduce the length of each reason, we implemented two types of `ConjunctionReducers`: `TruncationCReducer` that just truncates the conjunction reason to a specified length (*e.g.*, 7 conditionals), and `AttributeCReducer` that filters out from the conjunction all conditionals except those about a specified set of Widget Attributes. These attributes can be application-specific, being most salient to the application, most easily understood by users, or most privacy preserving. Reducers support requirement R5c by being extensible to support other heuristics.

Presenter

Even if the explanation is large, a developer may elegantly present it (*e.g.*, see Figure 7), instead of reducing it. Explainers produce explanations in the form of Explanation

Structs, and `Presenters` render them in a form presentable to end-users, *e.g.*, as text, visualization, or interactive graphical interface. Developers can build different Presenters to suit their target user and device form factor (requirement R5c).

We provide the reference implementation `RulesTextPresenter` that takes an Explanation Struct from the RulesExplainer and creates a text-based presentation based on templates (*e.g.*, see Figure 7). Explanations can also be presented to system components rather than to end-users. `EvidenceJsonPresenter` converts an Explanation Struct into JavaScript Object Notation (JSON) and posts it via a HTTP server call so that a remote client may consume the explanation.

A Presenter can also deal with combining explanations, such as presenting Certainty information with a Why explanation, showing Inputs for the What If explanation, or showing Output values for the Why Not and How To.

EXPLANATION GENERATION ALGORITHMS

We describe specific algorithms we developed to generate explanations from the four decision models that the Intelligibility Toolkit currently supports. Some explanation types are model-independent, and are supported by the generic Explainer. We describe the model-independent explanations first. *Inputs explanations* report the name and definition property of each context type (`Widget` attribute) used in the application. *Outputs explanations* report a `List` of output values for Enactor. *What explanations* report the current Enactor output value. To obtain input values (*Input What explanations*), handles for the input contexts are obtained via the Input explanation, and through those, the What explanation. A *What If explanation* sets input context values set by the user (through `UserInputsQuerier`), and tests it on all References. It reports the output value associated with the Reference that gets triggered. Model-specific explanations are generated from different Explainers for each model. Due to space constraints, our proofs are brief, but provide enough detail for replicability.

RulesExplainer

The Enactor framework supports one rule per Reference and we enforce one Reference per output. A decision model with multiple output values would have multiple rules, one per value. We make rules explainable by converting them into DNF by recursively applying De Morgan's Law, double negative elimination, and the distributive law. To explain how each explanation type is generated, we shall use the set up described in Table 1.

Why explanation

This explanation selects the rule(s) that was satisfied to produce the actual output (*e.g.*, see Figure 4, top). Note that multiple rules may be simultaneously satisfied.

Why Not explanation

This explanation selects rules that would achieve the target output value and, for each trace, identifies unsatisfied conditionals and returns a disjunction of traces containing these conditionals (Figure 4, bottom).

Input Conditionals	Output Values
a: Activity = Sitting ***b***: Noise = Quiet ***c***: Latitude *near* Office's ***d***: Longitude *near* Office's ***e***: Schedule = In Meeting	☺: Availability = Yes 😐: Availability = Somewhat Not ☹: Availability = Not

Table 1. Pedagogical example of input conditionals and output values for rule and decision tree. This describes an application to infer a user's availability based on his activity, the noise level around him, his proximity to his office (by latitude, longitude), and his schedule.

Input state (*a*, ¬*b*, ¬*c*, *d*, ¬*e*): user is sitting in a noisy place at latitude not near the office, longitude near the office, and is not in a meeting.

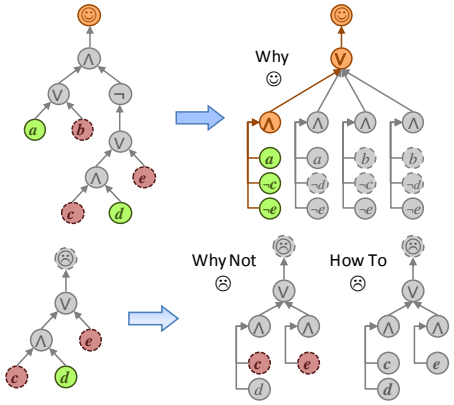

Figure 4: Generating explanations from Rules in DNF.

Why *available* (☺)? Because the user is sitting (*a*), is located at latitude *not* near his office (¬*c*), and is *not* in a meeting (¬*e*).

Why Not *unavailable* (☹)? Because he *isn't* near his office by latitude (*c*), or *isn't* in a meeting (*e*).

How To infer *unavailable* (☹)? He needs to be near his office by latitude (*c*) and longitude (*d*); or be in a meeting (*e*).

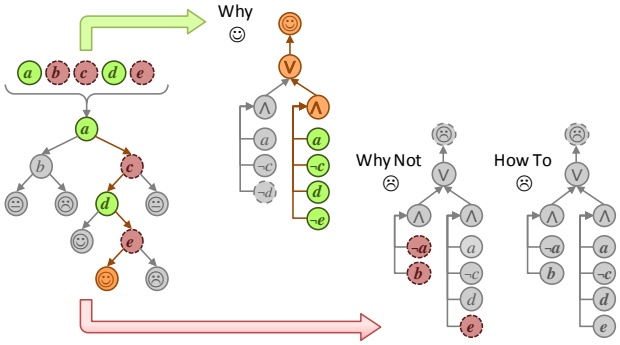

Figure 5: Generating from Decision Trees.

Why *available* (☺)? Because the user is sitting (*a*), is *not* located at latitude near his office (¬*c*), is located at longitude near his office (*d*), and is *not* in a meeting (¬*e*).

How To explanation

This explanation returns the DNF of the rule that achieves the target output, where each trace is a rule of how to achieve the target output value (Figure 4, bottom).

Certainty explanation

Enactor rules currently do not compute uncertainty, though uncertainty in inputs can be propagated descriptively.

J48Explainer (for Decision Tree)

This explainer generates explanations from the Weka [19] J48 implementation of the C4.5 decision tree [35]. There are some differences between decision trees and Enactor rules. Decisions for inferring the output are made from the top down, instead of from the bottom up as for rules; and decision trees encode traces for multiple output values within a single tree, unlike a rule-based structure. So DNF rules are created by traversing all paths in the tree and grouping paths by output values at their leaves. In DNF, we can generate explanations in the same way as for rules.

Why explanation

For every classification of an instance, the tree traces a single path to produce a Why explanation (Figure 5, top).

Why Not, and How To explanations

We retrieve the disjunction of traces that result in the target output value, then apply respective techniques as for rules.

Certainty explanation

Decision trees are built from statistical data, so they can model certainty from the probability distribution of remaining data points at each leaf.

NaiveBayesExplainer

To explain the naïve Bayes classifier, we extend the idea of *weights of evidence* demonstrated in [34] to multi-class problems. This approach explains additive classifiers by calculating an *evidence* score of how an instance is classified as a certain class value. If the score is positive, the classifier infers the class value, otherwise it infers another class value. The evidence can be decomposed linearly into its constituent evidences, indicating how much each feature contributes to the inference.

We start with the posterior probability that class $c \in C$ is inferred from a set of N class values given the observed instance feature input values f:

$$P(c|f) = P(c|f_1, f_2, \cdots, f_n) \propto P(c) \prod_{r=1}^{n} P(f_r|c)$$

where f_r is a feature of n possible values. The probability is calculated from the prior probability that a class would be in, $P(c)$, and the conditional probabilities of each feature value given the class, $P(f_r|c)$. c^i is inferred over other class values when $P(c^i|f) \geq P(c^j|f),\ \forall c^j \in C$. Since this is true for all class values, we can multiply all iterations of this inequality to get a combined expression:

$$\prod_{j=1}^{N} P(c^i|f) \geq \prod_{j=1}^{N} P(c^j|f).$$

Taking a logarithm gives us the *evidence* for the inference

$$g = log \prod_{j=1}^{N} P(c^i|f) - log \prod_{j=1}^{N} P(c^j|f) = g_i - g_{\sum} \geq 0$$

For brevity, we omit our working to derive the expression

$$g(c^i, r) = h + \left(\sum_{r=1}^{n} f\right) \qquad (1)$$

where $h(c^i) = N\ log\ P(c^i) - \mathcal{H}$ and $\mathcal{H} = \sum_{j=1}^{N} log\ P(c^j)$, $f(c^i, r) = N\ log\ P(f_r|c^j) - \mathcal{F}$ and $\mathcal{F} = \sum_{j=1}^{N} log\ P(f_r|c^j)$.

Figure 6: Why Not (Left), and How To + What If (Right) explanations for a mobile phone physical activity recognition application using accelerometer data trained with a naïve Bayes classifier. The application has inferred that the user is *Sitting*.

How to read Why Not: The top bar indicates the average evidence $\Delta g > 0$, *i.e.* more evidence for Sitting than Standing. The next bar indicates the evidence due to prior probabilities $\Delta h < 0$ is in favor of Standing (*i.e.* user is more likely to be standing). Each of the following bars indicates the difference in evidence for each feature, Δf, and whether they are in favor of Sitting or Standing.

Why Not inferred *Standing* but *Sitting*? Because (i) the prior likelihood indicates that it is pre-disposed to inferring Standing rather than Sitting, (ii) the values of Mean(x), Mean(y), Energy(x), Energy(y), *etc*, support the inference for Sitting, while (iii) the values of Mean(z), Energy(z), *etc*, support the inference for Standing. The user can interpret that the z-axis could be instrumental to infer Standing.

How To + What If explanation: User selects values of inputs, and see how the corresponding evidence changes along with the average overall evidence (bar at top), to see if the threshold (vertical bar) is crossed.

Naïve Bayes can be explained as the sum of evidence:

1. Prior probabilities of *selected class value*, $h(c^i)$
2. Due to each *feature value*, $f(c^i, r)$

See Figure 6 for an illustrative use of this explanation.

Why explanation

Why explanations are given in terms of weights of evidence of contributing factors and the total evidence, g. In this case, these factors are the predisposition, h, (prior probabilities) and the feature evidence, f, for each input.

Why Not explanations

Suppose the user is interested in an alternative target class value, $c^{i'}$, that was not inferred. We compute the evidence for the difference between the actual and target class values through a pairwise comparison, and knowing $P(c^i|f) > P(c^{i'}|f)$. The evidence of why c^i was inferred over $c^{i'}$ is

$$\Delta g(c^i, c^{i'}, r) = g(c^i, r) - g(c^{i'}, r) > 0 \qquad (2)$$

How To explanation

This shows weights of all features due to *normalized* values of the features, so the user can make sense of the general impact of each feature. (i) For nominal features, they only take a value of 0 or 1, so their evidence will either be 0 or *f*. (ii) For naïve Bayes, numeric features are commonly modeled by the Normal distribution; we "normalize" numeric features to a value of one standard deviation, $\sigma_{f|c^i}$, from the feature mean given the class value: $\mu_{f|c^i} \pm \sigma_{f|c^i}$.

In terms of what the user can do with input values to get a desired target output, if all values are nominal, we can permute all combinations and return those that achieve the target output. If multiple inputs are numeric, this becomes intractable. Instead, we could use a How To If explanation.

How To If explanation

This provides a tractable form of How To explanations (for nominal and numeric features) by constraining all feature values except one. For a numeric feature that is not fixed, we can vary the input value as it deviates *from the mean* and determine the threshold at which the outcome is achieved (or fails, if it is the opposite relation). A generalization of this with increased interactivity is the How To + What If explanation.

How To + What If explanation

One way to help users appreciate the influence of each weight is to allow users to speculate on the outcome with selected feature values. The What If explanation supports this, but does not necessarily start with sensible values to help users learn. The How To + What If explanation starts with the target class value, $c^{i'}$, and provides a set of *mean* feature values that satisfies this output, $\mu_{f|c^i}$. The user can then tweak the feature values to see if the target output value would still be inferred (Figure 6, right).

Certainty explanation

This reports the probability of inference, $P(c^i|f)$.

HMMExplainer (for Hidden Markov Model)

We apply the weights of evidence approach as used for naïve Bayes, additionally considering temporal factors. Once an HMM is learned (parameters π, A, B determined), inference of the state sequence is made by calculating its probability given an observation sequence

$$P(x^i|o) \propto P(x_1^i)\left(\prod_{t=2}^{T} P(x_t^i|x_{t-1}^i)\right)\left(\prod_{t=1}^{T} P(o_t|x_t^i)\right)$$

where $x_t^i \in X, i \in [1,N], t \in [1,T]$ is a state of N possible states at time t, in a sequence of length T, and o_t is the observation at that time. Taken together, the states represent a sequence $x^s, s \in [1,N^T]$. The probability is calculated from the prior probability that a state would be in, $P(x_1^i)$, the transition probabilities between the states, $P(x_t^i|x_{t-1}^i)$, and the emission probabilities of the observations given the states $P(o_t|x_t^i)$. For detailed information on HMMs, refer to [36]. While class and state refer to the same thing, we use the terminology consistent with HMM literature.

x^i is inferred over other states when $P(x^i|o) \geq P(x^s|o)$, $\forall x^s \in X$. Now, if we multiply this equation with all N^T permutations of s, we get $\prod_{s=1}^{N^T} P(x^i|o) \geq \prod_{s=1}^{N^T} P(x^s|o)$. Taking a logarithm gives us the *evidence* for the inference

$$\mathrm{g} = \log \prod_{s=1}^{N^T} P(x^i|o) - \log \prod_{s=1}^{N^T} P(x^s|o) = g_i - g_\Sigma \geq 0$$

At this juncture, we point out that we would like to present the evidence as a sum of feature evidences, instead of each observation as a whole. Viewing evidences in terms of full permutations of observations may be too difficult for end-users to assimilate. To do so, we make the *naïve assumption* (similarly used in naïve Bayes) that features are conditionally independent of one another given any state:

$$P(o_t|x_t^i) = P(o_{t_1}, o_{t_2}, \ldots, o_{t_n}|x_t^i) \propto \prod_{r=1}^{n} P(o_{t_r}|x_t^i).$$

Where o_{t_r} is the value of feature r of the observation at time t, and there are n features. With some working we get

$$\mathrm{g}(x^i, r, t) = \mathrm{h} + \left(\sum_{t=2}^{T} \mathrm{u}\right) + \left(\sum_{t=1}^{T} \sum_{r=1}^{n} \mathrm{f}\right) \qquad (3)$$

where

$$\mathrm{h}(x^i) = N^T \log P(x_1^i) - \mathcal{H}, \text{ and } \mathcal{H} = \left[\sum_{j=1}^{N} \log P(x_1^j)\right]^T,$$

$$\mathrm{u}(x^i, t) = N^T \log P(x_t^i|x_{t-1}^i) - \mathcal{U}, \text{ and}$$

$$\mathcal{U} = \left[\sum_{j1=1,j2=1}^{N,N} \log P(x_\tau^{j2}|x_{\tau-1}^{j1})\right]^{T-2},$$

$$\mathrm{f}(x^i, r, t) = N^T \log P(o_{t_r}|x_t^i) - \mathcal{F}, \text{ and}$$

$$\mathcal{F} = n^{T-1}\left[\sum_{j=1}^{N} \log P(o_{\tau_r}|x_\tau^j)\right]^T.$$

So, with the naïve assumption of independence among features, an HMM can be explained as the sum of evidence:

1. Prior probabilities of *selected state*, $\mathrm{h}(x^i)$
2. Due to each *state transition*, $\mathrm{u}(x^i, t)$
3. Due to each *feature value* at *sequence step*, $\mathrm{f}(x^i, r, t)$

We now have terms due to time, and the evidence for features are two-factored including time dependency. See Figure 8 for a demonstration of this explanation. Note that when $T = 1$, then we just have the naïve Bayes explanation.

Inputs and What If explanation

Though not formally an input, these explanations should also include state transitions. For the What If explanation, the user may want to speculate if a previous state was different, even though hidden states are actually inferred.

Why, Why Not, How To, and Certainty explanations

Same as for naïve Bayes, but with added evidence for time.

Reducing Dimensionality

If an application has many input features and/or a long sequence (*i.e*,. large n and/or large T), there may be too many evidence components. To reduce this dimensionality and make this more interpretable, we can sum evidence *by feature* across time (see Figure 8), or present evidence *by time* and sum evidence across features for each observation.

VALIDATION: DEMONSTRATION APPLICATIONS

To demonstrate that we can easily generate a range of explanations from context-aware applications using the Context Toolkit augmented with the Intelligibility Toolkit, we built three example intelligible applications. These examples demonstrate the use of the Intelligibility Toolkit for a span of *explanation types* and *model types*, and also cover a range of application domains for which the models are popular. While the toolkit significantly contributes to lowering the bar to providing explanations in context-aware applications, the explanations still need to be well designed (*e.g.*, for various UI, interaction, and device modality), and

Figure 7: IM Autostatus. Demonstration of various explanations from an IM responsiveness prediction plugin that uses a decision tree to predict when a buddy would respond.

crafted specifically for each application problem domain. Therefore, rather than examining different explanation methods for the same model or application (*e.g.*, [38]), we built different applications for each model type to *demonstrate the generality* of the Intelligibility Toolkit to provide explanations across application domains.

IM Autostatus Plugin – Rules / Decision Tree

Rule-based and decision tree-based explanations are similar so we only show decision tree explanations here. We built an AIM plugin that predicts when a buddy will respond to a message (Figure 7). It is trained on an existing dataset from [2] to build a decision tree. It takes desktop-based sensor inputs and makes response predictions (within/after 1 min).

We describe how we built this application in detail (the following two applications were built in a similar fashion) with the following procedure:

1. Create an ***Enactor*** for the overall application.
2. Create a ***Widget*** that tracks and updates all input features (extracted from the Subtle toolkit [14]).
3. Set a pre-trained J48 decision tree classifier model to be the Enactor's ***classifier***.
4. Set the Enactor's ***list*** of output values to two values: WITHIN_1_MIN and AFTER_1_MIN.
5. Create two ***References***, (a) associate them with the classifier and (b) associate one output value with one Reference. The developer implements what the application does when each Reference is triggered.
6. Create a ***RulesExplainer*** and associate with both References.
7. Set the DisjunctionReducer and ConjunctionReducer of the Explainer to ***FirstDReducer*** and ***TruncationCReducer***, respectively.
8. Create an IMAutostatusPresenter, a custom extension of ***RulesTextPresenter*** that understands what each feature means to provide domain-specific textual explanations. It also handles printing an AIM message.
9. Code UI elements to invoke, on user prompt, various getExplanation() functions from the Explainer. The corresponding ***Querier*** needs to be supplied when invoking each explanation type.

When the user asks for, say, a Why Not explanation about why not WITHIN_1_MIN:

1. The UI parses his request, populates an ***AltQuerier*** with WITHIN_1_MIN, and invokes ***getExplanation(WhyNot, AltQuerier)***.
2. The Enactor takes the returned explanation ***Expression*** and passes it to ***RulesTextPresenter*** that renders it for the user as an IM response.

Mobile Physical Activity Recognizer – Naïve Bayes

The naïve Bayes application we built is a physical activity recognizer that uses the accelerometer on a Google Android mobile phone to infer whether the user is sitting, standing, or walking (see Figure 6). It uses the ***NaiveBayesExplainer*** to generate all of the explanation types, ***FirstDReducer*** and ***TruncationCReducer*** to simplify the explanations, and ***EvidenceJsonPresenter*** to render the explanation in a server call. The Android application retrieves the explanation and renders the bar chart visualization.

Home Activity Recognizer – HMM

We demonstrate explanations from an HMM model using the dataset from [21] about domestic activity, and train a HMM with a sequence length of 5 min, and 1 min per sequence step. The application takes 14 binary input sensors and infers which activity (out of 7) the user is performing. It uses an instance of ***HMMExplainer*** to generate all of the explanation types, ***FirstDReducer*** and ***TruncationCReducer*** to simplify the explanations, and a custom ***Presenter*** to exaggerate and visualize the explanation evidences (see Figure 8).

LIMITATIONS AND DISCUSSIONS

While the Intelligibility Toolkit is extensible, the current implementation does not cover some outstanding aspects. (i) It does not cover the less-frequently used model types (*e.g.*, SVM, clustering) and does not handle ensemble techniques of composite classifiers (*e.g.*, bagging, boosting). (ii) There remain some types of explanations that users ask for that are not yet supported: *e.g.*, how to Control an application to change its behavior (though this is supported with Parameters in the Enactor framework [13]), History (when something happened/changed, trends), and Provenance (source, credibility, and accuracy of inputs). (iii) Often, sensed raw inputs of context-aware applications are pre-processed, *e.g.*, using signal processing or computer vision techniques. Although important [33], the toolkit does not currently capture and explain these pre-

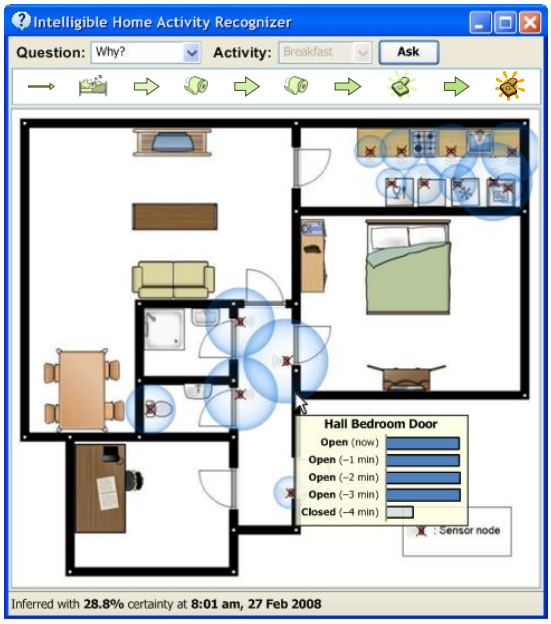

Figure 8: Demonstration of Why explanation visualization from an application using a HMM to model domestic activity.

This explains why the application inferred a sequence of Sleeping → Toilet → Toilet → Breakfast → Breakfast in the last 5 min. Evidence due to features (summed across the last 5 min) are indicated by the area of bubbles around the corresponding sensors in the floorplan. Evidence for each sensor across time is revealed in a tooltip.

We can see that the Hall Bedroom Door being open is a strong indicator of inferring the sequence. The door being open is a stronger indicator than it being closed 4 min ago. The microwave is another strong indicator (biggest bubble in top right corner).

processing mechanisms. (iv) Applications may encounter behaviors due to the infrastructure rather than the decision model or inputs. For example, unexpected behavior may result from resources suddenly being unavailable due to connectivity issues. The toolkit does not currently support explanations about the infrastructure.

RELATED WORK IN EXPLAINING CONTEXT

The Intelligibility Toolkit supports a wide range of explanations for multiple decision models. Previous systems only covered a subset of the explanation types, and only for one or one type of decision model.

The most similar framework to our toolkit is the Enactor framework [13], on which we base our Intelligibility Toolkit. It can provide What explanations by exposing the state of input Widgets, and Why explanations by reporting a relevant rule. However, it does not support the other explanation types or any models beyond rules. The Crystal framework [30] supports only Why / Why Not explanations for desktop-based applications to explain themselves through Command Objects. The Whyline [22] similarly explains Why / Why Not to end-user programmers, by examining the program execution tree. PersonisAD [1] defines a distributed framework to support What explanations by resolving identities and associations of devices, locations, people, *etc*. The Intelligent Office System [10] provides What explanations by showing the system state, History explanations by listing the states across time, and Why explanations about the learned cut-points for its rules. Panoramic [44] provides Why, What, and History explanations to explain location events through a visualization of parallel timelines of sensed and rule-determined events.

While the aforementioned systems provide explanations for rules, Tullio *et al*. [42] explained interruptibility inferred from decision trees and naïve Bayes with What explanations. The Intelligibility Toolkit can provide deeper (*e.g.*, Why, Why Not, How To) explanations from these models. Kulezsa *et al*. [24] built an intelligible email sorter that uses naïve Bayes for classification. It provides Why, Why Not, and What If explanations based on the weights of evidence approach [34]. The Intelligibility Toolkit also uses this approach, and adds more explanation types, supports numeric input features, extends it for HMMs, and has been developed to be extensible.

CONCLUSION AND FUTURE WORK

We have presented the Intelligibility Toolkit that currently provides automatic generation of eight explanation types (Inputs, Outputs, What, What If, Why, Why Not, How To, Certainty) for the four most popular decision model types (rules, decision trees, naïve Bayes, hidden Markov models) in context-aware applications. It supports the generation of explanation structures (through Explainers), querying mechanisms to specify questions and constrain explanations (through Queriers), simplifying complex explanations (through Reducers), and presenting the explanations to end-users and other subsystems (through Presenters). The toolkit is also extensible to support new explanation types, model types, reduction heuristics, and presentation formats. The Intelligibility Toolkit makes it easier for developers to provide many explanation types in their context-aware applications. This ease also allows developers to perform rapid prototyping of different explanation types to discern the best explanations to use and the best ways to use them.

In addition to addressing the limitations outlined earlier, we will use the Intelligibility Toolkit, to pursue further research questions regarding the intelligibility of context-aware applications. In particular, we can investigate and compare the efficacy of various explanation types, by measuring how well each type helps users to understand the application, and improve their trust in the application. We also plan to deploy an intelligible application with multiple explanation types, and multiple context types (*e.g.*, location, physical activity), and conduct a longitudinal evaluation of the impact of intelligibility and how well it improves understanding or corrects misunderstanding.

ACKNOWLEDGMENTS

This work was funded by the National Science Foundation under grant 0746428, and the Agency for Science Technology And Research, Singapore. We also thank Stephanie Rosenthal, Somchaya Liemhetcharat, Jen Mankoff, Zhiquan Yeo, Andreas Möller, Brian Ziebart, Matt L. Lee, and Bryan Pendleton for their helpful advice.

REFERENCES

1. Assad, M. *et al*. (2007). PersonisAD: Distributed, Active, Scrutable Model Framework for Context-Aware Services. *Pervasive 07*, 55-72.
2. Avrahami, D. & Hudson, S.E. (2006). Responsiveness in Instant Messaging: Predictive Models Supporting Inter-Personal Communication. *CHI 06*, 731-740.
3. Bardram, J. E. (2005). The Java Context Awareness Framework (JCAF) – A Service Infrastructure and Programming Framework for Context-Aware Applications. *Pervasive 05*, 98-115.
4. Barkhuus, L. & Dey, A.K. (2003). Is context-aware computing taking control away from the user? Three levels of interactivity examined. *Ubicomp 03*, 149–156.
5. Bellotti, V. & Edwards, W.K. (2001). Intelligibility and Accountability: Human Considerations in Context-Aware Systems, *Human-Computer Interaction*, 16(2-4): 193-212.
6. Borriello, G. *et al.* (2004). Reminding About Tagged Objects Using Passive RFIDs. *Ubicomp 04*, 36-53.
7. Bulling, A., Ward, J. A., Gellersen, H., & Tröster, G. (2008). Robust Recognition of Reading Activity in Transit Using Wearable Electrooculography. *Pervasive 08*, 19-37.
8. Chang, K., Hightower, J., & Kveton, B. (2009). Inferring Identity Using Accelerometers in Television Remote Controls. *Pervasive 09*, 151-167.
9. Chang, K., Chen, M. Y., & Canny, J. (2007). Tracking Free-Weight Exercises. *Ubicomp 09*, 19-37.
10. Cheverst, K.. *et al.* (2005). Exploring issues of user model transparency and proactive behavior in an office environment control system. *UMUAI 05*, 15(3-4), 235-273.
11. Dey, A.K., Abowd, G.D. & Salber, D. (2001). A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction*, 16(2–4): 97–166.
12. Dey, A. K. & de Guzman, E. (2006). From awareness to connectedness: the design and deployment of presence displays. *CHI 06*, 899-908.
13. Dey, A. K. & Newberger, A. (2009). Support for context-aware intelligibility and control. *CHI 09*, 859-868.
14. Fogarty, J. & Hudson, S. E. (2007). Toolkit support for developing and deploying sensor-based statistical models of human situations. *CHI 07*, 135-144.
15. Franois, J.M. (2010). Jahmm: An implementation of Hidden Markov Models in Java. http://code.google.com/p/jahmm/. Retrieved 9 Mar 2010.
16. Froehlich, J. *et al.* (2009). UbiGreen: investigating a mobile tool for tracking and supporting green transportation habits. *CHI 09*, 1043-1052.
17. Gregor, S. & Benbasat, I. (1999). Explanations From Intelligent Systems: Theoretical Foundations and Implications for Practice. *MIS Quarterly* 23(4): 497–530.
18. Gu, T., Pung, H. K., & Zhang, D. Q. (2005). A service-oriented middleware for building context-aware services. *Journal of Network and Computer Applications*, 28(1), 1-18.
19. Hall, M. *et al*. (2009). The WEKA Data Mining Software: An Update. *SIGKDD Explorations 09*, 11(1), 10-18.
20. Herlocker, J., Konstan, J. & Riedl, J. (2000). Explaining collaborative filtering recommendations. *CSCW 00*, 241-250.
21. Kasteren, T. L. M. *et al.* (2008). Accurate Activity Recognition in a Home Setting. *Ubicomp 08*, 1-9.
22. Ko, A. J. & Myers, B. A. (2009). Finding causes of program output with the Java Whyline. *CHI 09*, 1569-1578.
23. Kofod-Petersen, A., & Mikalsen, M. (2005). Context: Representation and reasoning – Representing and reasoning about context in a mobile environment. *Revue d'Intelligence Artificielle*, 19, 479–498.
24. Kuleza, T. *et al*. (2009). Fixing the Program My Computer Learned: Barriers for End-users, Challenges for the Machine. *IUI 09*, 187-196.
25. Lim, B. Y., Dey, A. K. (2009). Assessing Demand for Intelligibility in Context-Aware Applications. *Ubicomp 09*, 195-204.
26. Lim, B. Y., Dey, A. K. & Avrahami, D. (2009). Why and why not explanations improve the intelligibility of context-aware intelligent systems. *CHI 09*, 2119-2128.
27. McGuinness, D. *et al.* (2007). A Categorization of Explanation Questions for Task Processing Systems. *AAAI Workshop on Explanation-Aware Computing (ExaCt-07)*.
28. Mozina M. *et al.* (2004). Nomograms for Visualization of Naive Bayesian Classifier. *PKDD 2004*, 337-348.
29. Muir, B. (1994). Trust in automation: Part i. theoretical issues in the study of trust and human intervention in automated systems. *Ergonomics*, 37(11): 1905–1922.
30. Myers, B. A. *et al.* (2006). Answering why and why not questions in user interfaces. *CHI 06*, 397-406.
31. Newcomb, E., Pashley, T., & Stasko, J. (2003). Mobile computing in the retail arena. *CHI 03*, 337-344.
32. Olsen, D. R. (2007). Evaluating user interface systems research. *UIST 07*, 251-258.
33. Patel, K. *et al.* (2008). Investigating Statistical Machine Learning as a Tool for Software Development. *CHI 08*, 667-676.
34. Poulin, B. *et al.* (2006). Visual explanation of evidence in additive classifiers. *IAAI 06*, 1822-1829.
35. Quinlan, J. R. (1993). C4.5: Programs for Machine Learning. Morgan Kaufmann Publishers.
36. Rabiner L. R. (1989). A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.
37. Robnik-Šikonja, M. & Kononenko, I. (2008). Explaining Classifications For Individual Instances. IEEE *Transactions on Knowledge and Data Engineering*, 20(5): 589-600.
38. Stumpf, S. *et al.* (2009). Interacting meaningfully with machine learning systems: Three experiments. *International Journal of Human-Computer Studies*, 67(8), 639-662.
39. Tapia, E. M., Intille, S. S., Larson, K. (2004). Activity Recognition in the Home Using Simple and Ubiquitous Sensors. *Pervasive 04*, 158-175.
40. Terada, T. *et al.* (2004). Ubiquitous Chip: A Rule-Based I/O Control Device for Ubiquitous Computing. *Pervasive 04*, 238-253.
41. Tsukada, K. & Yasumura, M. (2004). ActiveBelt: Belt-Type Wearable Tactile Display for Directional Navigation. *Ubicomp 04*, 384-399.
42. Tullio, J. *et al.* (2007). How it works: A field study of non-technical users interacting with an intelligent system. *CHI 07*, 31-40.
43. Weiser, M. & Brown, J. S. (1997). The coming age of calm technology. *Beyond Calculation: the Next Fifty Years*, 75-85.
44. Welbourne, E., Balazinska, M., Borriello, G., Fogarty, J. (2010). Specification and Verification of Complex Location Events. *Pervasive 10*, 57-75.
45. Zheng, Y. *et al.* (2008). Understanding mobility based on GPS data. *Ubicomp 08*, 312-321.