Learning Based Java for Rapid Development of NLP Systems

Nick Rizzolo, Dan Roth

University of Illinois at Urbana-Champaign
{rizzolo,danr}@illinois.edu

Abstract

Today's natural language processing systems are growing more complex with the need to incorporate a wider range of language resources and more sophisticated statistical methods. In many cases, it is necessary to learn a component with input that includes the predictions of other learned components or to assign simultaneously the values that would be assigned by multiple components with an expressive, data dependent structure among them. As a result, the design of systems with multiple learning components is inevitably quite technically complex, and implementations of conceptually simple NLP systems can be time consuming and prone to error. Our new modeling language, Learning Based Java (LBJ), facilitates the rapid development of systems that learn and perform inference. LBJ has already been used to build state of the art NLP systems. This paper details recent advancements in the language which generalize its computational model, making a wider class of algorithms available.

1. Introduction

As the fields of Natural Language Processing (NLP) and Computational Linguistics have matured, more sophisticated language resources and tools have become available. These tools perform complicated analyses of natural language text to find named entities, identify the argument structure of verbs, determine the referents of pronouns and nominal phrases, and more. Many such tasks involve multiple learning components whose collective objective is to assign values to variables that may have an expressive, data dependent structure among them. Thus, systems that perform these tasks have complicated, data dependent development cycles and run-time interactions. As such, their implementations become large and unwieldy, which can restrict their usefulness as resources.

Organized infrastructure solutions such as GATE (Cunningham et al., 2002), NLTK (Loper and Bird, 2002), and IBM's UIMA (Götz and Suhre, 2004) only partially solve these issues. They aim to make separately learned components "plug-and-play", but they do not help manage their training nor do they offer solutions when the outputs of different components contradict each other. The more recently developed *Alchemy* (the most popular MLN (Richardson and Domingos, 2006) implementation) and FACTORIE (McCallum et al., 2009) systems offer general purpose solutions for global training and inference, but they lack the flexibility to decompose the problem, and general purpose algorithms quickly become intractable on the large problems encountered in NLP.

A comprehensive solution for modeling problems in NLP (as well as other domains) would combine the advantages of both types of systems mentioned above. It would make effortless the combination of arbitrary types of components in the learned system, be they learned or hard coded (e.g. features and constraints). At the same time, it would allow the modeling of large, structured problems over which learning and inference can be performed globally. However, in contrast to the systems above, it should also allow a flexible decomposition of such large, structured problems so that learning and inference can be efficiently tailored to suit the problem.

We refer to the whole of these principles as Learning Based Programming (LBP) (Roth, 2006). Our previous work introduced Learning Based Java[1] (LBJ) (Rizzolo and Roth, 2007), a modeling langauge that represented a first step in this direction. It modeled a user's program as a collection of locally defined experts whose decisions are combined to make them globally coherent. While this is certainly one type of decomposition LBP aims to provide, the language lacked the expressivity to specify other interesting models.

This paper makes three main contributions. First, we demonstrate that there exists a theoretical model that describes most, if not all, NLP approaches adeptly (Section 2.). Second, we describe our improvements to the LBJ language and show that they enable the programmer to describe the theoretical model succinctly (Sections 3. and 4.). Third, we introduce the concept of *data driven compilation*, a translation process in which the efficiency of the generated code benefits from the data given as input to the learning algorithms (Section 5.). Thus, the programmer spends his time designing his models instead of worrying about the low level details of writing efficient learning based programs that have been abstracted away.

2. A Model for NLP Systems

We submit the *constrained conditional model* (CCM) of (Chang et al., 2008) as the paradigmatic NLP modeling framework. A CCM can be represented by two weight vectors, $\mathbf{w}$ and ρ, a set of feature functions $\Phi = \{\phi_i \,|\, \phi_i : \mathcal{X} \times \mathcal{Y} \rightarrow \mathbb{R}\}$, and a set of constraints $\mathcal{C} = \{C_j \,|\, C_j : \mathcal{X} \times \mathcal{Y} \rightarrow \mathbb{R}\}$. Here, $\mathcal{X}$ is referred to as the *input space* and $\mathcal{Y}$ is referred to as the *output space*. Most often, both are multi-dimensional. Let $\mathfrak{X}$ be the set of possible values for a single element of the input, and let Υ be similarly defined for the output. Then $\mathcal{X} = \mathfrak{X}^p$ and $\mathcal{Y} = \Upsilon^q$ for integers p and q.

The score for an assignment to the output variables $\mathbf{y} \in \mathcal{Y}$ on an input instance $\mathbf{x} \in \mathcal{X}$ can then be obtained via the

[1] Java is a registered trademark of Sun Microsystems, Inc.

linear objective function

$$f(\mathbf{x}, \mathbf{y}) = \sum_i w_i \phi_i(\mathbf{x}, \mathbf{y}) - \sum_j \rho_j C_j(\mathbf{x}, \mathbf{y}), \quad (1)$$

and inference is performed by selecting (perhaps approximately) the highest scoring output variable assignment:

$$\mathbf{y}^* = \operatorname*{argmax}_{\mathbf{y} \in \mathcal{Y}} f(\mathbf{x}, \mathbf{y}) \quad (2)$$

While features and constraints are defined to return real values above, they are often Boolean functions that return 0 or 1 in this context. The only difference between them is that a feature's weights are set by a learning algorithm, whereas a constraint's weights are set by a domain expert. Thus, constraints are a mechanism for incorporating knowledge into the model. Note that CCMs are not restricted to any particular learning or inference algorithms. Thus, the designer of the model can tailor the semantics of the features and weights for the task at hand.

The CCM is very general and subsumes many modeling formalisms. As such, many, if not all models developed in the NLP community fall under its umbrella. For the rest of this section, we will explore these claims in more depth.

2.1. Classical Models of Learning

The simplest types of models are predictors for discrete variables. CCM is also general enough to model real valued variables, but regression is rarely utilized in NLP, so we will omit that discussion here. Below, we consider some familiar learning models that can all be realized as CCMs. They are all unconstrained, so the second summation in equation (1) can be ignored for now.

2.1.1. Linear Threshold Units

Binary classification algorithms such as Perceptron (Rosenblatt, 1958) and Winnow (Littlestone, 1988) represent their hypothesis with a weight vector $\mathbf{w}$ whose dimensions correspond to features of the input $\{\phi'_i(\mathbf{x})\}$. The prediction of the model is then $y^* = sign(\mathbf{w} \cdot \Phi'(\mathbf{x}))$, the dot product between the weight vector and the features is compared with a threshold $\theta = 0$. Thus, we refer to these models as linear threshold units (LTUs).

To cast this model as a CCM, we first note that $\Upsilon = \{-1, 1\}$ and $\mathcal{Y} = \Upsilon$. There are no restrictions on $\mathfrak{X}$ or $\mathcal{X}$. Then we simply distribute the output variable y into the definitions of the features:

$$\phi_i(\mathbf{x}, y) = y\, \phi'_i(\mathbf{x}) \quad (3)$$

Equations (1) and (2) can then be used for inference. All w_i and $\phi'_i(\mathbf{x})$ are fixed, so the objective function remains linear.

2.1.2. Multi-Class Classifiers

A popular approach to online multi-class classification instantiates for each class a separate LTU $\mathbf{w}_y$, $y \in \Upsilon$, indexed by the same features of the input $\{\phi'_i(\mathbf{x})\}$ (Carlson et al., 1999; Crammer and Singer, 2003). The prediction is then simply the class associated with the highest scoring weight vector $y^* = \operatorname{argmax}_{y \in \mathcal{Y}} \mathbf{w}_y \cdot \Phi'(\mathbf{x})$.

Once again, to cast this model as a CCM, we have $\mathcal{Y} = \Upsilon$, and we distribute the output variable into the definitions of the features. However, in this case, valid values $\hat{y} \in \Upsilon$ of the output variable will also be used to index the features (Punyakanok et al., 2005):

$$I_{\hat{y}}(y) = \begin{cases} 1 & \text{if } y = \hat{y} \\ 0 & \text{otherwise} \end{cases} \quad (4)$$

$$\phi_{i,\hat{y}}(\mathbf{x}, y) = I_{\hat{y}}(y)\, \phi'_i(\mathbf{x}) \quad (5)$$

$$f(\mathbf{x}, y) = \sum_{i,\hat{y}} w_{i,\hat{y}} \phi_{i,\hat{y}}(\mathbf{x}, y) \quad (6)$$

Equation (4) effectively redefines our output space from a single, discrete variable into a set of Boolean variables. Equation (6) simply shows the objective function f from equation (1) with the new feature indexing scheme. It is linear in the new $I_{\hat{y}}(y)$ variables, and we can use equation (2) for inference.

Generative models used for multi-class classification such as naïve Bayes can also be viewed in this light (Roth, 1999).

2.1.3. Hidden Markov Models

The standard in sequential prediction tasks is the Hidden Markov Model (HMM) (Rabiner, 1989). It is a generative model that incorporates (1) a probability of making each possible emission at step i and (2) a probability of being in each possible state at step $i + 1$, both conditioned on the state at step i. These probabilities are usually organized into emission and transition probability tables, $P(e_i|s_i)$ and $P(s_{i+1}|s_i)$, respectively, where $s_i \in \mathcal{S}$ and $e_i \in \mathfrak{E}$. During inference, the emissions e_i are fixed, the state variables s_i are our output variables, and our goal is to find the assignment that maximizes likelihood or, equivalently, log-likelihood:

$$\mathbf{s}^* = \operatorname*{argmax}_{\mathbf{s}} \prod_{i=1}^{n} P(s_i|s_{i-1}) P(e_i|s_i) \quad (7)$$

$$= \operatorname*{argmax}_{\mathbf{s}} \sum_{i=1}^{n} \log(P(s_i|s_{i-1})) + \log(P(e_i|s_i)) \quad (8)$$

where s_0 is a special 0^{th} state symbol placed at the beginning of every sequence.

Following (Collins, 2002), we can cast equation (8) as a CCM by first flattening the log probabilities into our weight vector. Next, we rearrange equation (8) to factor out the model's weights, which are just the individual probabilities in the two tables:

$$I_{\hat{r},\hat{r}'}(r, r') = I_{\hat{r}}(r) I_{\hat{r}'}(r') \quad (9)$$

$$\mathbf{s}^* = \operatorname*{argmax}_{\mathbf{s}} \sum_{\hat{s},\hat{e}} \log(P(\hat{e}|\hat{s})) \left(\sum_{i=1}^{n} I_{\hat{s},\hat{e}}(s_i, e_i) \right) + \sum_{\hat{s},\hat{s}'} \log(P(\hat{s}|\hat{s}')) \left(\sum_{i=1}^{n} I_{\hat{s},\hat{s}'}(s_i, s_{i-1}) \right) \quad (10)$$

It is now clear that our features simply count the number of occurrences of each (*state, emission*) pair and each pair of consecutive states in the sequence. Thus, with $\mathcal{X} = \mathfrak{C}^n$ and $\mathcal{Y} = \mathcal{S}^n$, we can complete our CCM definition as follows:

$$\phi_{\hat{x},\hat{y}}(\mathbf{x},\mathbf{y}) = \sum_{i=1}^{n} I_{\hat{x},\hat{y}}(x_i, y_i) \tag{11}$$

$$\phi_{\hat{y},\hat{y}'}(\mathbf{x},\mathbf{y}) = \sum_{i=1}^{n} I_{\hat{y},\hat{y}'}(y_i, y_{i-1}) \tag{12}$$

$$f(\mathbf{x},\mathbf{y}) = \sum_{\hat{x},\hat{y}} w_{\hat{x},\hat{y}} \phi_{\hat{x},\hat{y}}(\mathbf{x},\mathbf{y}) + \sum_{\hat{y},\hat{y}'} w_{\hat{y},\hat{y}'} \phi_{\hat{y},\hat{y}'}(\mathbf{x},\mathbf{y}) \tag{13}$$

Our objective function (13) is once again linear in the variables $I_{\hat{x},\hat{y}}(x_i, y_i)$ and $I_{\hat{y},\hat{y}'}(y_i, y_{i-1})$. As Collins notes, we can then solve equation (2) efficiently with the Viterbi algorithm.

2.2. Multivariate NLP Models

In recent years, NLP systems have moved away from models of single output variables to incorporate many decisions simultaneously. But these joint models must still be decomposed to be tractable during both learning and inference. Thus, many researchers now use classical models as building blocks for the decomposition of their systems. They use constraints to encode structural relationships between these building blocks as well as prior knowledge about their global behavior. Additionally, they frequently infuse further knowledge into the system by controlling the behavior of the inference algorithm. CCMs can accommodate all of these modeling techniques.

A prime example of this modeling philosophy is the semantic role labeling (SRL) system of (Punyakanok et al., 2008). In SRL, the input $\mathbf{x}$ represents a sentence of natural language text. The sentence must be segmented into phrases which may represent arguments of a given verb in the sentence. Each phrase that does represent an argument must be classified by its type. While a solution to this problem could be learned in a joint probabilistic framework, Punyakanok, et al. decomposed it into two independently learned components and hard constraints encoding prior knowledge enforced only at inference time. They showed that this decomposition resulted in more efficient learning requiring less training data as well as a fast inference strategy. We now discuss the implementation of this system as a CCM.

Decomposition: Their system accepted an array $\mathbf{x}$ of n argument candidates as input. They learned, independently, one linear threshold unit to act as an argument candidate filter, and one multi-class classifier to predict argument types. Both classifiers classify a single argument candidate $x \in \mathbf{x}$ and were trained with features of only the input $\Phi'_F(x)$ and $\Phi'_T(x)$, respectively. The filter predicts either `yes` or `no`. The type classifier selects a prediction from $\mathcal{T} \cup \{\texttt{null}\}$ where $\mathcal{T}$ is the set of argument types (e.g. `A0`, `A1`, `A2`, ...) and `null` indicates the candidate argument is not actually an argument. So, the CCM will include two output variables $y_{j,F} \in \{-1, 1\}$ and $y_{j,T} \in \mathcal{T} \cup \{\texttt{null}\}$ for each argument candidate x_j. We can write its feature functions as follows.

$$\phi_{i,F}(\mathbf{x},\mathbf{y}) = \sum_{j=1}^{n} y_{j,F}\, \phi'_{i,F}(x_j) \tag{14}$$

$$\phi_{i,\hat{y},T}(\mathbf{x},\mathbf{y}) = \sum_{j=1}^{n} I_{\hat{y}}(y_{j,T})\, \phi'_{i,T}(x_j) \tag{15}$$

Constraints: If the filter predicts `no`, the type classifier must predict `null`. We will refer to this structural constraint as the *filter constraint*. In addition, there are the structural constraints ensuring that no two arguments overlap as well as knowledge about type regularities encoded in constraints such as

- no two arguments associated with any given verb may have type `A`*t*, for $t \in \{0, 1, 2, 3, 4, 5\}$, and
- if any argument associated with a verb v has reference type `R-A`*t*, then some other argument associated with v must have the referent type `A`*t*, for $t \in \{0, 1, 2, 3, 4, 5\}$.

Constraints were defined at the beginning of this section as returning a real value, just like features. However, they are often most useful as new Boolean output variables $C(\mathbf{x},\mathbf{y}) \in \{0, 1\}$ indicating whether some desirable property of the other variables has been violated. In this case, their definition often comes in the form of linear inequalities. Here is the linear definition of the filter constraint:

$$C_{j,F}(\mathbf{x},\mathbf{y}) \geq I_{-1}(y_{j,F}) - I_{\texttt{null}}(y_{j,T}) \tag{16a}$$

$$2C_{j,F}(\mathbf{x},\mathbf{y}) \leq I_{-1}(y_{j,F}) - I_{\texttt{null}}(y_{j,T}) + 1 \tag{16b}$$

The inequalities (16) establish that $C_{j,F}(\mathbf{x},\mathbf{y})$ will be 1 if the type variable for argument x_j is non-`null` when its filter variable says `no` (i.e., the filter constraint has been violated), and 0 otherwise. Unlike our feature definitions, these inequalities must reside outside the objective function as separate constraints on the inference problem.

Constraints that establish a logical relationship between output variables can be written to enforce the other structural and domain specific constraints in our SRL problem as well (Punyakanok et al., 2008). In fact, any constraint written in a logical form can be translated to such linear inequalities automatically (Rizzolo and Roth, 2007). We omit the descriptions of the remaining constraints for lack of space.

Inference: The inference strategy employed by Punyakanok, et al. was motivated by empirical evidence they gathered indicating that a prediction of `no` from the filter was correct a high percentage of the time. As such, they chose to trust these decisions more than decisions made by the type classifier. This behavior can be implemented in a CCM by artificially inflating the filter's scores by a constant α.

$$f(\mathbf{x},\mathbf{y}) = \alpha\, \mathbf{w}_F \cdot \Phi_F(\mathbf{x},\mathbf{y}) + \mathbf{w}_T \cdot \Phi_T(\mathbf{x},\mathbf{y}) - \infty \mathbf{C}(\mathbf{x},\mathbf{y}) \tag{17}$$

This will cause the model to prefer, in general, global assignments that agree with the filter classifier. Note also that the constraints are all *hard*; ie., if any constraint is violated, the score of the assignment is $-\infty$.

```
1.  model ArgumentIdentifier :: discrete[] input -> boolean isArgument
2.     input[*] /\ ^isArgument;
3.  model ArgumentType :: discrete[] input -> discrete type
4.     input[*] /\ type;
5.     input[*] /\ input[*] /\ type;
6.  static model pertinentData :: ArgumentCandidate candidate
7.                             -> discrete[] data
8.     data.phraseType = candidate.phraseType();
9.     data.headWord   = candidate.headWord();
10.    data.headTag    = candidate.headTag();
11.    data.path       = candidate.path();
```

Figure 1: The SRL system from Section 2.2. is decomposed into two learned components whose general structure is defined in lines 1-5. Lines 6-11 define a hard-coded model that collects data from a Java object for later use as input variables for the learned components.

2.3. Other CCMs in the Wild

Examples of more complicated CCMs abound in the NLP literature. (Barzilay and Lapata, 2006) describes an automatic semantic aggregator that uses constraints to control the number of aggregated sentences and their lengths. (Marciniak and Strube, 2005) describes a general constraint framework for solving multiple NLP problems simultaneously. (Martins et al., 2009) describes a dependency parsing system that incorporates prior knowledge as hard constraints. These and other systems would be more easily maintainable, more portable, and more useful as resources if they had been developed in a modeling formalism designed specifically for them. We aim to provide such an environment in Learning Based Java.

3. Learning Based Java

Learning Based Java has already been used to develop several state-of-the-art resources. The LBJ POS tagger[2] reports a competitive 96.6% accuracy on the standard Wall Street Journal corpus. In the named entity recognizer of (Ratinov and Roth, 2009), non-local features, gazetteers, and wikipedia are all incorporated into a system that achieves 90.8 F_1 on the CoNLL-2003 dataset, the highest score we are aware of. Finally, the co-reference resolution system of (Bengtson and Roth, 2008) achieves state-of-the-art performance on the ACE 2004 dataset while employing only a single learned classifier and a single constraint.

Nevertheless, our previous work on LBJ was not expressive enough to represent features involving multiple output variables. This paper redesigns LBJ to represent, learn, and perform inference over arbitrary CCMs. We introduce our modeling language by example. The codes in Figures 1, 2, and 3 specify the structure of the Punyakanok, et al. semantic role labeling system.[3] These figures discuss how LBJ language constructs address the concerns of the SRL system as described in Section 2.2. Section 3.1. discusses each in turn. Section 3.2. then describes the syntax of features and constraints in more detail.

[2] http://L2R.cs.uiuc.edu/∼cogcomp/software.php

[3] Some of the features and constraints have been omitted to save space.

3.1. Models

A model in LBJ simply represents an objective function of the form of equation (1) in which the weights $\mathbf{w}$ are implicit (recall that ρ is specified by a human; thus it is explicit). Features and constraints are specified in a logic syntax as described in Section 3.2. Once these are specified, the model can be instantiated so that each instance contains its own weight vectors.

Decomposition: Figure 1 immediately describes the unit of decomposition used to build the system. The two models declared on lines 1 and 3 are the models that will do all the system's learning. The `ArgumentIdentifier` model will be a linear threshold unit, so it has a `boolean` output variable. Its body declares features in the form of equation (3). The `ArgumentType` model will be a multi-class classifier, so it has a `discrete` output variable. Its features are declared in the form of equation (5). (The syntax for writing these features on lines 2, 4, and 5 is described in Section 3.2.) Finally, Figure 1 declares a model used merely to extract the data we wish to utilize in these learned models. We will see in Figure 3 how this data is given to them.

In more detail, a model declaration's header contains a name for the model and a list of argument specifications. The list is partitioned by an arrow (`->`) indicating that the arguments on the left represent input, and the arguments on the right represent output variables. Input may mean input variables, primitive types, or Java objects from the programmer's main program. The variables (either input or output) in these examples are the ones with types `boolean` or `discrete`. They are intended precisely to represent the $\mathbf{x}$ and $\mathbf{y}$ input and output variables in equation (1).

Any model may be declared *static*, and it has roughly the same meaning as the same keyword when used on a Java method. Models with no learnable parameters are usually declared static. A model may also be *hard-coded*, though there is no keyword for this property. A hard-coded model is one whose output is well defined even without learning any parameters. The `pertinentData` model on line 6 which contains only assignment statements is both static and hard-coded.

Constraints: Figure 2 contains the implementations for some of the constraints in this SRL system. The first model

```
1.  static model noOverlaps :: ArgumentCandidate[] candidates -> discrete[] types
2.     for (i : (0 .. candidates.size() - 1))
3.        for (j : (i + 1 .. candidates.size() - 1))
4.           #: candidates[i].overlapsWith(candidates[j])
5.              => types[i] :: "null" || types[j] :: "null";
6.  static model noDuplicates :: -> discrete[] types
7.     #: forall (v : types[0].values)
8.        atmost 1 of (t : types) t :: v;
9.  static model referenceConsistency :: -> discrete[] types
10.    #: forall (value : types[0].values)
11.       (exists (var : types) var :: "R-" + value)
12.          => (exists (var : types) var :: value);
```

Figure 2: Structural constraints and domain specific expert knowledge encoded as hard constraints are defined here as separate models with no learning components.

```
1.  model SRLProblem :: ArgumentIdentifier ai, ArgumentType at,
2.                      ArgumentCandidate[] candidates
3.                   -> boolean[] isArgument, discrete[] types
4.     for (i : (0 .. candidates.size() - 1))
5.        100: isArgument[i] <- ai (pertinentData candidates[i]);
6.        1:   types[i]      <- at (pertinentData candidates[i]);
7.        #:   ~isArgument[i] => types[i] :: "null";
8.     types[*] <- noOverlaps candidates;
9.     types[*] <- noDuplicates ();
10.    types[*] <- referenceConsistency ();
```

Figure 3: The SRL system from Section 2.2. This code captures the decomposition of the inference problem into two learned components and several hard constraints. A wide variety of learning and inference approaches can now be applied over this structure.

declares a structural constraint over every pair of argument candidates that says, "if two constraints overlap, they can't both have non-`null` type." The other two models encode knowledge about the global behaviors of the output variables. The model on line 6 says, "each argument type may appear at most once in the sentence," and the model on line 9 says, "no reference type may appear unless it corresponds to a referent."

None of these models contain any learned weights. However, they are not considered hard-coded because there are usually multiple valid outputs they might produce for a given input, and LBJ makes no guarantee as to which will be chosen.

Inference: Figure 3 puts all these components together in the global model. By applying the learned models on each argument candidate (as in lines 5 and 6), we construct the objective function in equation (17). The scaling factors appear at the beginning of the lines before the colon and give preference to the decisions of the filter classifier. This results in features of the form described by equations (14) and (15). We also enforce the filter constraint as defined by equations (16) on line 7. Finally, the externally defined constraints are applied to the global model in lines 8-10.

The `SRLProblem` model takes native Java objects representing candidate SRL arguments as input, defines input and output variables with respect to those objects, and *applies* the learned models and hard constraints over those variables. Model application is a mechanism through which the relationships described by one model can be established amongst selected variables in another model. It is accomplished by *binding* the inputs and outputs of the applied model to the inputs and outputs (respectively) in the *parent* model (which is `SRLProblem` in this case). Binding is different than assignment, because no results have been computed. Instead, we are simply declaring that a particular externally defined model's structure appears in this model.

Lines 5, 6, 8, 9, and 10 of Figure 3 are all examples of model application. They use the left arrow (`<-`) operator, which binds variables in the manner described above. The model application itself appears to its right, and the newly bound output variables appear to its left. In general, inputs must be bound with inputs and outputs with outputs. The only exception is when the applied model is hard-coded. Since a hard-coded model's output is already completely determined and cannot be affected by the context in which it is applied, its output variables may be bound with input variables in a parent model. We see `pertinentData` applied in this way in lines 5 and 6.

Finally, lines 8-10 enforce hard constraints in the model. Unlike the model applications on lines 5 and 6, these three model applications each bind to the entire dictionary of type variables. In order for this type of binding to make sense, the indexes (which can be integers or strings) used to access the dictionary must be consistent from the applied model to the parent model. In this case, for example, we used the same integers in both contexts.

3.2. Features and Constraints

The relationships between variables that we have alluded to throughout this paper come in the form of *features* and *constraints*. The roles that these two constructs play in a CCM are very similar. Each one simultaneously

- distinguishes a potential property of the variables and
- measures the presence or *strength* of that property in the variables' current values.

They are specified as predicates in a first order logic (FOL) syntax in which variables play the role of objects. That syntax contains the usual connectives and quantifiers, as well as equality and inequality predicates (`::` and `!:` respectively) and quantifiers that can compare the quantities of objects that satisfy a predicate.

3.2.1. Features

To distinguish features from each other, we give them *names*. These names act as the indexes on features, constraints, and weight vectors we saw in Section 2. The strength of the feature is the real valued result that is multiplied by the weight with the same name. However, for simplicity, in this paper we will focus on Boolean features, whose strengths can be 0 or 1.

Features' names and values come from the variables they are functions of, and the structure of the feature functions themselves. As in most programming languages, variables are referred to with identifiers and are used to store interesting bits of data. LBJ has *Boolean*, *discrete*, and *real* variables. It also provides dictionaries in which the keys act as separate variable names. Dictionaries can be accessed with either integers or strings inside square brackets (e.g. `tags["foo"]`) or the selection operator and an identifier (e.g. `tags.foo`). Both of those syntax examples will refer to the same variable.

Anywhere in the body of a model, a declarative feature statement indicates that the model will include a weight associated with the specified feature. For example, `headWord :: "office"` is a feature that evaluates to *true* if and only if the `headWord` variable takes the value `"office"`. The name of the feature will be its entire lexical form, after any interpolation that need be done in the indexes of dictionaries.

3.2.2. Constraints

The same FOL syntax is available for the specification of constraints, except that each constraint statement is prefixed with a real-valued literal or a # symbol standing for ∞ followed by a colon. This value is the ρ corresponding to the constraint in the objective function. It represents the *penalty* that is incurred iff the constraint is *violated*.

Constraints tend to make more frequent use of the quantifiers `forall`, `exists`, `atleast`, `atmost`, and `exactly`. These quantifiers have essentially the same semantics as in LBJ's prior version, though the `exactly` quantifier is new. Its form is:

`exactly` *n* `of` (*var* : *set*) *sentence*

and it is semantically equivalent to

`atleast` *n* `of` (*var* : *set*) *sentence*
`/\ atmost` *n* `of` (*var* : *set*) *sentence*

3.2.3. Extensions to First Order Logic

LBJ extends the typical semantics of these logical sentences to allow several unorthodox types of atoms. First (and least ground-breaking), `boolean` variables may appear in a feature statement anywhere an atom normally would. They are treated as if they are 0-ary predicates.

Second, discrete variables may appear as atoms. Whenever discrete variables appear as atoms without being compared to a value or another variable via the `::` or `!:` operators, the feature statement that contains them actually represents many features, one for each set of values in the cross product of the variables in question. Each of these features will have its own weight in the model. For example, given a discrete variable `A` $\in$ { `"a1"`, `"a2"`, `"a3"` } and a Boolean variable `B`, the feature `A /\ B` represents three separate features: `A :: "a1" /\ B`, `A :: "a2" /\ B`, and `A :: "a3" /\ B`.

Third, a dictionary with an index of `*` (e.g. `types[*]`) may appear as an atom. When it does, each variable in the dictionary is substituted into the feature in turn, each creating a new feature statement. These new feature statements are subject to the same rules described above depending on whether the substituted variables are Boolean or discrete.[4]

Thus, it is easy to specify the CCM structure of a multi-class classifier as we saw in Section 3.1. Special provisions must be made to accomodate this behavior with logical operators other than conjunction (Cumby and Roth, 2003). LBJ currently does not make those provisions, but the same effects are possible with quantifiers and equality predicates.

New to this version of LBJ is the `boolean` variable, which is an atomic feature. There is also a new operator for manipulating Boolean features (though we only envision it useful when applied to `boolean` output variables) denoted `^`, an example of which appears on line 2 of Figure 1. This operator changes the range of its argument from $\{0, 1\}$ to $\{-1, 1\}$, thereby making it possible to model linear threshold units as described in Section 2.1.1.

4. Learning and Inference

So far, we have shown how to define the shape and structure of a Constrained Conditional Model using Learning Based Java. The code we have written so far defines that structure and nothing more; it is completely agnostic to both learning and inference. From here, with the help of a sufficiently comprehensive library, the average programmer should need only select the algorithms of his choice.

For inference in particular, one of the key advantages to implementing a model as a CCM is that it is always possible to fall back on Integer Linear Programming (ILP) to solve the inference problem. Since CCMs keep their objective

[4] Constraints typically do not make use of discrete variables or dictionaries as atoms, but if they do, all resulting constraints are given the same value for their ρ.

```
1.  for (i : (1 .. vars.size()-1))
2.     newScores <- double[];
3.     for (current : vars[i].values)
4.        for (prev : vars[i-1].values)
5.           s = scores[prev] + problem.score {vars[i] = current; vars[i-1] = prev};
6.           if s > newScores[current] then
7.              newScores[current] = s;
8.              predictions[i][current] = prev;
9.     scores = newScores;
```

Figure 4: A code sample from an LBJ implementation of the Viterbi algorithm. On line 5, the model returns the score of a partial assignment to the output variables. This is a computational building block for many inference algorithms.

functions linear and their features and constraints in a logic language, they can be automatically translated to ILP optimization problems. While ILP is intractable in general, it has been successful in practice on a variety of tasks, even when incorporating long range constraints (Punyakanok et al., 2008; Denis and Baldridge, 2007; Martins et al., 2009). However, if the task at hand demands a more problem specific approach, LBJ can help.

4.1. Inference

Inference in LBJ is often as simple as naming the algorithm and the output variables to apply it to. This is the case in the following code, where we see the implementation for an approximate solution to our running SRL example.

```
1. solver SRLInference :: SRLProblem problem
2.    Greedy.solve problem.isArgument[*];
3.    ILP.solve    problem.types[*];
```

First it applies a greedy algorithm to the `isArgument` variables, literally executing the argmax in equation (2) over each output variable individually. The resulting assignments for these variables are now fixed, making the job of the next inference algorithm called a little easier.

However, it is often the case that the structure of the problem indicates a particularly appropriate algorithm that was not anticipated in the LBJ library. For example, the HMM (Section 2.1.3.) is efficiently solved by Viterbi. Of course, LBJ has a Viterbi implementation, and Figure 4 shows a snippet from it. But from this snippet, we can see an important bit of LBJ's syntactic sugar that makes writing inference algorithms easier. On line 5, the model is queried for the score of a partial assignment to the output variables.

A partial assignment score query can be performed over any subset of output variables. The result is the usual evaluation of equation (1), except every feature and constraint function whose evaluation depends on a variable outside the partial assignment is assumed to return 0. In the context of a CCM specified in LBJ, the programmer also has access to the names of the variables and can thereby pick out their structure to guide his inference procedure. Thus, a host of ad hoc inference implementations become possible.

4.2. Learning

Like inference algorithms, learning algorithms can be implemented externally and linked to LBJ. However, LBJ also provides several facilities that make it easier to write learning algorithms. First, output variables can contain both labels and predicted values. This comes in handy when writing a supervised learning algorithm. Second, a model can act as a feature extractor that returns a *feature vector*. Features can be extracted using either the labels or the current predicted values in the output variables. Third, the language contains syntactic sugar that lets models be treated as weight vectors for the purpose of performing linear algebra with respect to feature vectors. Combined with the ability to query for the scores of partial assignments as described above, the programmer has the necessary tools for building custom learning solutions quickly.

5. Data Driven Compilation

The biggest advantage to developing a machine learning framework as a stand-alone language as opposed to a library for an existing general purpose langauge is that it opens many opportunities for automatically improving the efficiency of the code based on high level analyses. LBJ exploits these analyses with a unique twist, since much of the information necessary to generate the final program code is only available in the training data. Thus, we say that an LBJ compiler performs *data-driven compilation*. Feature extraction is perhaps the biggest beneficiary of data driven compilation.

In most NLP systems, a lexicon associating each feature with a unique integer index is built from the training data. These integers are used to index the weight vector, which is implemented simply as an array. Many NLP systems create a separate entry in the lexicon's hash table for every unique feature. Since many NLP systems have millions of features, the resulting code will use a lot of memory and will be slowed by the abundance of accesses to the hash table.

Lexicons created by LBJ, on the other hand, only store indexes associated with the discrete values each input and output variable are observed to take. For any discrete variable that can take one of k possible values, each value is associated with a number between 0 and $k - 1$ inclusive in the lexicon. Then they organize the feature index space so that features that have the same topology while merely comparing their constituent variables with different values are grouped together. This will happen frequently, since features that use discrete variables and dictionaries as atoms are quite common.

Under this organized feature index space, we can now com-

pute recursively, as functions of the indexes of their subexpressions, the indexes of the larger formulas that are active given a variable assignment. These indexing functions get their behavior from the connectives used in the feature formulae. For example, if a feature f is a conjunction of two formulae f_1 and f_2, its active indexes will take the form $\mathcal{I}(f) = k_{f_2}\mathcal{I}(f_1) + \mathcal{I}(f_2) + \Omega_f$, where k_{f_2} is the number of features in the same group as f that differ only by value comparisons made in f_2, and Ω_f is an offset that ensures the index space of f begins immediately after the previous feature's index space ended.

Disjunction complicates things a little, since many features in a group of disjunctive features can be active simultaneously. For example, when the features `A :: "foo" \/ B :: "bar"` and `A :: "foo" \/ B :: "baz"` are grouped together, both will be active if the variable `A` is set to `"foo"`. The result is that sets of active indexes are returned up the recursion, and the parent formula's index computation loops over the cross product of these sets to compute its indexes.

The constants in the index formulae can be computed at compile time, after an initial pass over the data, but before training begins. The end result is a lexicon orders of magnitude smaller and generated code that performs swift feature extraction, making any algorithm implemented in the language more efficient.

6. Conclusion

In this paper we described a modeling formalism for multivariate models (CCM) and showed that it is appropriate for a wide variety of NLP tasks. We then developed a programming language (LBJ) for specifying the models and peforming learning and inference over them. Finally, we showed that the feature extraction syntax of the language can be compiled to code efficient in both space and time. Using LBJ, we believe NLP systems that use learning and inference can be developed rapidly, since the developer will spend most of his time thinking about the modeling of his problem from a high level.

Acknowledgements

The authors would like to thank Ming-Wei Chang, James Clarke, and Vivek Srikumar for many insightful conversations. This work was supported by NSF grant NSF SoD-HCER-0613885.

7. References

R. Barzilay and M. Lapata. 2006. Aggregation via Set Partitioning for Natural Language Generation. In *Proc. of HLT/NAACL*.

E. Bengtson and D. Roth. 2008. Understanding the Value of Features for Coreference Resolution. In *Proc. of EMNLP*.

A. Carlson, C. Cumby, J. Rosen, and D. Roth. 1999. The SNoW Learning Architecture. Technical report, UIUC Computer Science Department.

M. Chang, L. Ratinov, N. Rizzolo, and D. Roth. 2008. Learning and Inference with Constraints. In *Proc. of AAAI*.

M. Collins. 2002. Discriminative Training Methods for Hidden Markov Models: Theory and Experiments with Perceptron Algorithms. In *Proc. of EMNLP*.

K. Crammer and Y. Singer. 2003. Ultraconservative Online Algorithms for Multiclass Problems. *Journal of Machine Learning Research*.

C. Cumby and D. Roth. 2003. Feature Extraction Languages for Propositionalized Relational Learning. In *IJCAI Workshop on Learning Statistical Models from Relational Data*.

H. Cunningham, D. Maynard, K. Bontcheva, and V. Tablan. 2002. GATE: A Framework and Graphical Development Environment for Robust NLP Tools and Applications. In *Proc. of ACL*.

P. Denis and J. Baldridge. 2007. Joint Determination of Anaphoricity and Coreference Resolution using Integer Programming. In *Proc. of NAACL*.

T. Götz and O. Suhre. 2004. Design and Implementation of the UIMA Common Analysis System. *IBM Systems Journal*.

N. Littlestone. 1988. Learning Quickly When Irrelevant Attributes Abound: A New Linear-threshold Algorithm. *Machine Learning*.

E. Loper and S. Bird. 2002. NLTK: the Natural Language Toolkit. In *Proceedings of the ACL-02 Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics*.

T. Marciniak and M. Strube. 2005. Beyond the Pipeline: Discrete Optimization in NLP. In *Proc. of CoNLL*.

Andre Martins, Noah Smith, and Eric Xing. 2009. Concise Integer Linear Programming Formulations for Dependency Parsing. In *Proc. of ACL*.

A. McCallum, K. Schultz, and S. Singh. 2009. FACTORIE: Probabilistic Programming via Imperatively Defined Factor Graphs. In *NIPS*.

V. Punyakanok, D. Roth, W. Yih, and D. Zimak. 2005. Learning and Inference over Constrained Output. In *Proc. of IJCAI*.

V. Punyakanok, D. Roth, and W. Yih. 2008. The Importance of Syntactic Parsing and Inference in Semantic Role Labeling. *Computational Linguistics*.

L. R. Rabiner. 1989. A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition. *Proceedings of the IEEE*.

L. Ratinov and D. Roth. 2009. Design Challenges and Misconceptions in Named Entity Recognition. In *Proc. of CoNLL*.

M. Richardson and P. Domingos. 2006. Markov Logic Networks. *Machine Learning Journal*.

N. Rizzolo and D. Roth. 2007. Modeling Discriminative Global Inference. In *Proc. of ICSC*.

F. Rosenblatt. 1958. The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. *Psych. Rev.* (Reprinted in *Neurocomputing* (MIT Press, 1988).).

D. Roth. 1999. Learning in Natural Language. In *Proc. of IJCAI*.

D. Roth. 2006. Learning Based Programming. *Innovations in Machine Learning: Theory and Applications*.