Appeared in ICSC 2007

Modeling Discriminative Global Inference

Nicholas Rizzolo and Dan Roth
University of Illinois at Urbana-Champaign
Cognitive Computations Group
201 N. Goodwin Ave., Urbana, IL 61801
{rizzolo,danr}@uiuc.edu

Abstract

Many recent advances in complex domains such as Natural Language Processing (NLP) have taken a discriminative approach in conjunction with the global application of structural and domain specific constraints. We introduce LBJ, a new modeling language for specifying exact inference systems of this type, combining ideas from machine learning, optimization, First Order Logic (FOL), and Object Oriented Programming (OOP). Expressive constraints are specified declaratively as arbitrary FOL formulas over functions and objects. The language's run-time library translates them to a mathematical programming representation from which an exact solution is computed. In addition, the compiler leverages an existing OOP language: objects and functions are grounded as the OOP objects and methods that encapsulate the user's data.[1]

1. Introduction

Engineering complexities inherent in the design of systems based on generative models have spurred the development of several useful modeling languages. The Integrated Bayesian Agent Language (IBAL) [7] is a rational, probabilistic modeling language. A program represents a situation that an agent has encountered and the actions it can take, and in evaluating the program, IBAL uses probabilistic inference to induce the described models and determine what a rational agent would do. The PRogramming In Statistical Modeling (PRISM) language [12] is a probabilistic modeling language modeled after Prolog. It is designed for inference with the EM learning algorithm. More recently, the Bayesian LOGic language (BLOG) [5] allows the user to define generative models in terms of varying numbers of unknown objects. Complete inference algorithms are provided. Finally, [10] introduces Markov Logic Networks (MLNs), a combination of First Order Logic (FOL) with probabilistic graphical models. MLNs also create a generative model and come with inference algorithms.

Modeling languages such as these ease the burden of the programmer and have led to many recent advances in complex domains such as Natural Language Processing (NLP). However, they often restrict the user to a particular representation or inference algorithm, and their inference often cannot be solved exactly.

Moreover, generative models are not the only actively studied formalism. Discriminative models are often preferred because they allow greater flexibility in the design of the model and its inference algorithms as well as the efficiency necessary for those algorithms to produce exact solutions. This paper's thesis is that discriminative systems can also benefit from a modeling language that abstracts away implementation details, making systems easier to program and less error prone.

We introduce Learning Based Java (LBJ), a new modeling language for discriminative systems, designed for use with the Java(TM)[2] programming language. An LBJ program models the decision making functionality of a system as a collection of locally defined experts whose decisions are combined to make them globally coherent. First, the problem's discrete functions (the local experts) are identified, and probability distributions are imposed over their possible values. Often, they are induced from data automatically with learning algorithms. Then, structural and domain specific constraints specified as arbitrary FOL formulas are enforced to ensure global consistency while maximizing a function of their values' probabilities. We refer to this process as inference, and we use Integer Linear Programming (ILP) to compute an exact solution.

ILP has been a successful problem solving paradigm in practice, despite the fact that it is intractable in the limit. In particular, it has been shown an effective technique for combining the output of many local decision makers into a coherent, exact, global inference. [1] describes an automatic semantic aggregator that uses constraints to control the number of aggregated sentences and their lengths.

[1]This research is supported by NSF grant SoD-HCER-0613885.

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

[9] and [8] describe semantic role labeling systems that incorporate both structural and domain specific constraints. [4] describes a general ILP framework for solving multiple NLP problems simultaneously.

The rest of this paper is organized as follows. In Section 2 we formalize the mathematical model whose implementation LBJ aims to facilitate. Section 3 then introduces LBJ by describing some examples of its use to model NLP problems. Next, we formalize the syntax and semantics of the language in Section 4. In Section 5, we show how arbitrary FOL formulas can be translated to ILP constraints. We discuss the merits of our system in practice in Section 6, and we finally conclude in Section 7.

2. A Discriminative Inference Model

An LBJ program models the interaction between discrete functions as a constrained optimization problem. Discrete functions in LBJ are similar to FOL functions in that their application to a tuple of objects represents another object. However, LBJ functions are all unary, and they each return a single string.

Definition 1. *An LBJ function $f \in \mathcal{F}$ is defined as $f : \mathbb{O} \rightarrow \mathbb{S} \cup \{\texttt{null}\}$, where $\mathbb{O}$ is the set of all objects and $\mathbb{S}$ is the set of all strings. A function f may be partial when its domain is a subset of $\mathbb{O}$. When a function is applied to an object outside its domain, the* `null` *value is returned. Each f is associated with a scoring function denoted $g_f : \mathbb{O} \times \mathbb{S} \rightarrow \mathbb{R}$.*

An LBJ function f (and its associated g_f) may be specified explicitly or learned from data in terms of other functions.

Discriminative inference can now be modeled as a constrained optimization problem as follows. Let $\mathcal{F}_p \subset \mathcal{F}$ be the set of discrete functions involved in an instance of our inference problem. Let $\mathbb{O}_i \subset \mathbb{O}$ be the set of objects $o_{i,j}$ on which $f_i \in \mathcal{F}_p$ is evaluated in the inference. Let $\mathbb{S}_i \subset \mathbb{S}$ be the set of values $s_{i,k}$ that f_i is capable of returning given an object. Then we can express linearly an objective function that represents the expected number of correct predictions:

$$z(\mathbf{I}) \equiv \sum_{i=1}^{|\mathcal{F}_p|} \sum_{j=1}^{|\mathbb{O}_i|} \sum_{k=1}^{|\mathbb{S}_i|} g_{f_i}(o_{i,j}, s_{i,k}) I_{i,j,k} \qquad (1)$$

where $I_{i,j,k} \in \{0, 1\}$ is an indicator variable set to 1 to indicate that $f_i(o_{i,j}) = s_{i,k}$ and 0 otherwise. Our goal will be to find the values of the $I_{i,j,k}$ variables that maximize this objective function while satisfying certain constraints. First, we immediately include a set of constraints implied by the nature of our functions:

$$\sum_{k=1}^{|\mathbb{S}_i|} I_{i,j,k} = 1 \quad \forall i, j \qquad (2)$$

Structural, domain, and task specific constraints are then added, encoding the interactions between functions' applications to objects on an object by object basis and ensuring a coherent, global solution is produced.

So far, we have specified our optimization problem with linear, arithmetic formulas, and we must specify all constraints in the same fashion if we wish to utilize the highly optimized linear programming codes that exist today. However, many interesting constraints are quite difficult to express as linear inequalities. LBJ offers First Order Logic as an alternative.

Definition 2. *An LBJ constraint $c \in \mathcal{C}$ is an FOL formula in which the predicates are equality comparisons between the outputs of functions applied to objects and/or strings.*

While the model as a whole may seem restrictive at first, linear models are in fact quite flexible. Many complex models can be expressed linearly - even structured models with multiple outputs [3, 11]. Plus, as we will show in Section 5, any constraint written as a Boolean formula can be expressed as a set of linear inequalities. With such expressivity, the promise of an exact solution, and efficiency in practice [4, 8], LBJ's discriminative inference model solved with an algorithm such as ILP becomes a natural choice. It should also be noted that while we prefer ILP since it yields an exact solution, the framework does not preclude heuristic approaches such as beam search.

In practice, several engineering issues arise when implementing this model. First, the design of an application that uses learned functions typically becomes fragmented across the various third party codes used to perform feature extraction and learning. Evaluation of learned functions is almost always performed off-line, before code that uses those results begins to run. The syntax and semantics of LBJ functions are designed to integrate feature extraction and learning in the programmer's application. In addition, the programmer need not change internal representations, since the objects and strings in the domain and range of each LBJ function are grounded as the programmer's Java objects and strings. This makes the application's code conceptually cleaner, more portable, and easier to maintain while simultaneously eliminating (when possible) the use of inefficient disk accesses to communicate intermediate results.

Furthermore, third party codes need only be wrapped in Java classes that extend the appropriate LBJ library classes to be accessible to the LBJ programmer. Thus, learning algorithm implementations such as the very useful WEKA library [13] can all benefit immediately from discriminative inference. One small caveat is that since WEKA's implementations are not sparse (as those in LBJ's library are), the programmer must take care when using WEKA to ensure that only those features that appeared during training appear during evaluation.

```
1.  discrete POSTagger(Word w) cachedin w.partOfSpeech <-
2.  learn POSLabel
3.    using { return w.spelling; }, baselineTarget, labelTwoBefore,
4.          labelOneBefore, labelOneAfter, labelTwoAfter, L2bL1b, L1bL1a, L1aL2a
5.    from new POSBracketParser("data/POS/corpus.train")
6.    with new SparseNetworkLearner(new SparsePerceptron(.1, 0, 2))
7.  end
```

Figure 1. Snippet from a state of the art POS tagger (97.4% accuracy) coded in 54 lines of LBJ.

```
1.  inference ChunkingInference head Sentence sentence {
2.    Chunk c { return c.getSentence(); }   // returns the "head" object
3.    normalizedby Softmax;   // converts scores to distributions
4.    subjectto {
5.      forall (Word w in sentence.allWords())
6.        atmost 1 of (Chunk c in sentence.chunksContaining(w))
7.          LocalChunkType(c) !: null;   // "!:" means not equal
8.    }
9.    with new GLPK()   // names the inference algorithm
10. }
11. discrete ChunkType(Chunk c) <- ChunkingInference(LocalChunkType)
```

Figure 2. Snippet from an LBJ program for simple chunker inference. The `subjectto` clause defines a structural constraint: `LocalChunkType` should only label non-overlapping chunks. `ChunkType` (line 11) will respect this constraint.

Constraints (and inference procedures) typically cause system designs to fragment in the same way that learned functions do. They must be generated on the fly, since the elements of each $\mathbb{O}_i$ and even the elements of $\mathcal{F}_p$ may not be known until the data is observed. Most often, task specific code is written to generate the constraints' representations in the language of some third-party constraint solver. This is performed as an extra off-line stage after the learned functions are evaluated on test data and before the application runs. LBJ provides a convenient, general purpose constraint language and automatically generates the code that instantiates those constraints given the data's objects. It then provides the programmer with a new set of LBJ functions that make globally consistent decisions on-line.

3. Examples

In the previous section, we introduced the mathematical model behind LBJ. In the next section, we will delve into the technical details of the language itself. This section provides a bridge between the two by examining how three NLP tasks are modeled and coded in LBJ.

3.1. Part of speech (POS) tagging

The POS tagging task yields a very simple instantiation of our model. Any given word represents an inference problem p for which the set $\mathcal{F}_p$ contains only a single function f_1: the POS tagger itself, as acquired from a learning algorithm applied to training data. Associated with that function are the sets $\mathbb{O}_1$ containing only the given word and $\mathbb{S}_1$ containing all possible parts of speech. Finally, we need not specify any constraints beyond (2) (the implied constraints), meaning that the objective function (1) can be maximized simply by picking each part of speech that maximizes the score returned by the POS tagger. Thus, LBJ need not invoke any inference algorithm (such as ILP) for this task.

The code snippet in Figure 1 specifies a POS tagger that achieves 97.4% accuracy on the WSJ corpus from the Penn TreeBank. It illustrates how a function can be defined in terms of a learning algorithm, data, and other functions used as feature extractors. In this case, the function `POSTagger` takes an object of class `Word` as input and returns a `discrete` value (i.e., a string; while the language also allows a `real` return value, we limit our discussion to the discrete case in this paper). While its parameterized form is known at programming-time (it is implied by the learning algorithm), its parameters will not be known until the data is processed (at compile-time), and they may continue to change via on-line learning at run-time.

3.2. Chunking

The code snippet in Figure 2 specifies a simple "chunker" that partitions a sentence into groups of contiguous words that have some well defined syntactic or semantic functionality. For example, "chunks" may be classified with a type such as "NP" for noun phrase or "VP" for verb

```
1.  constraint LegalArguments(SRLSentence sentence) {
2.    for (Iterator I = sentence.candidates.iterator(); I.hasNext(); ) {
3.      LinkedList forVerb = (LinkedList) I.next();
4.      String verb = ((Argument) forVerb.getFirst()).verb.lemmaForm;
5.      LinkedList legal = PropBankFrames.getLegalArguments(verb);
6.      forall (Argument a in forVerb)
7.        exists (String type in legal) ArgumentClassifier(a) :: type; } }
8.  constraint NoA0Duplicates(SRLSentence sentence) {
9.    forall (LinkedList forVerb in sentence.candidates)
10.     atmost 1 of (Argument a in forVerb)
11.       ArgumentClassifier(a) :: "A0"; }   // "::" means equal
```

Figure 3. Snippet from an LBJ program for Semantic Role Labeling. The constraints defined encode useful domain knowledge that makes the global classification coherent.

phrase. Not every word in the sentence need be included in a chunk, but the words in any two labeled chunks must be mutually exclusive (non-overlapping). In this simple example, our approach will be to develop a discrete function that is capable of making such a type prediction about any set of contiguous of words from a given sentence.

Our model is instantiated for this task by first learning a single multi-class classifier `LocalChunkType` to predict the type of a chunk (the code that specified the learning has been omitted for brevity). A sentence will then represent an inference problem. The set $\mathcal{F}_p$ will always contain only `LocalChunkType`, while $\mathbb{O}_1$ contains every possible chunk in the given sentence (i.e., every sequence of words) and $\mathbb{S}_1$ contains all possible chunk types. Unlike the POS tagging task, we must now add constraints to our inference problem, beyond those that are implied, to ensure that no two overlapping chunks are each given a non-`null` label. LBJ will then invoke an inference algorithm (such as ILP) which, in effect, modifies our functions' outputs so that they maximize (1) while respecting the constraints.

In LBJ, we specify our constraint using the `inference` syntax exhibited in Figure 2. From its header, we see that an `inference` has a `head` object from which all example objects involved in the constraints are accessible. In line 2, a simple clause similar to a Java method body directs the run-time inference mechanism to the `head` object corresponding to the given function input object. Line 3 specifies an algorithm for normalizing the scores produced by `LocalChunkType` so that they may be treated as a discrete distribution. Finally, the `subjectto` clause (line 4) contains the code that imposes our structural constraint.

The `subjectto` clause in Figure 2 only contains a single, declarative constraint statement expressing the following idea. For every word w in the sentence, there may exist no more than one chunk containing w for which `LocalChunkType` predicts something other than `null`. Thus, when `ChunkingInference` is applied to the `LocalChunkType` function in line 11, the result is a new function `ChunkType` which never labels both of two overlapping chunks as non-`null`.

3.3. Semantic Role Labeling (SRL)

SRL is a large scale task similar to chunking in that the goal is to predict partitionings of the given sentence and to label each partition. However, in SRL, we must create one such partitioning for each verb in the sentence. The partitions are then referred to as *arguments* and labeled according to their function with respect to the corresponding verb. (See [2] for a more detailed description of the SRL task.) Thus, a sentence-verb pair represents an inference problem. Several learned functions (some that classify words and one that classifies arguments) participate in the inference, so that $|\mathcal{F}_p| > 1$ and each $\mathbb{O}_i$ contains different objects. Plus, in addition to the structural constraints that are similar to chunking, systems such as [8] encode domain specific knowledge as constraints. The code in Figure 3 gives examples of this.

In Figure 3, we see two examples of the `constraint` declaration. The `LegalArguments` constraint ensures that each argument type prediction is a valid argument type for the corresponding verb as determined with the PropBank Frames [6]. The `NoA0Duplicates` constraint ensures that no more than one argument corresponding to any given verb will be labeled "A0". Note that both of these constraints are encodings of prior knowledge about the domain, and that the latter is often difficult or impossible to enforce when using certain inference algorithms such as dynamic programming and certain forms of beam search. This is because the constraint simultaneously affects multiple parts of the sentence that may be far apart from each other [11].

4. A Discriminative Modeling Language

In this section, we describe in more detail the syntax and semantics of the language. An LBJ program can be formalized as a tuple, $\mathcal{P} = \langle \mathcal{F}, \mathcal{C}, \mathcal{I} \rangle$, where $\mathcal{F}$ is a set of LBJ function declarations, $\mathcal{C}$ is a set of LBJ constraint declarations, and $\mathcal{I}$ is a set of LBJ inference declarations.

4.1. Functions

In our LBJ implementation, functions need not be defined in the same source file as they are used. In fact, they need not be defined in any LBJ source file. The only prerequisite to referencing a function is that a Java class with the same name as the function is accessible to the LBJ compiler and that it extends the appropriate class in the LBJ run-time library, implementing the abstract methods. When a function is defined in the source file, the LBJ compiler will automatically create an appropriate Java class for it.

The syntax of a function declaration names the function and specifies its input and output types in its header, which is similar to a Java method header. The header is followed by a left arrow indicating assignment and then by the *function expression* (Section 4.2) to be assigned to the name.

Semantically, every named function is a Java method returning a `Feature` object. Objects of class `Feature` store two strings representing the feature's name, which is usually just the name of the function that produced it, and value. LBJ ensures that as learning algorithms create the representations of the functions they learn, unique strings are stored exactly once in memory. Each named function's definition is stored as a static method in a Java class of the same name, such that the method may be accessed through objects of the class. The class also provides a method that directly returns the value of the produced `Feature`. Thus, the programmer rarely works directly with class `Feature` in the application and never in the LBJ source; it serves mainly to manage the values computed by functions behind the scenes.

4.2. Function Expressions

An LBJ function expression is either a name, a method body, the conjunction of two function expressions, or a learning function expression. Each of these options is discussed in turn in this section. "Evaluation" of a function expression (done at compile-time) results in an anonymous function, similar to a functional programming language.

The name of a function defined either externally or in the LBJ source file may appear wherever a function expression is expected. Functions need not be defined before being referenced. If the named function is declared in another Java package, it must either be fully qualified (e.g., `myPackage.myFunction`) or it must be imported by an import declaration at the top of the source file. The Java files containing the implementations of each imported function must exist prior to running the LBJ compiler on the source file that imports them.

A method body is a list of Java statements enclosed in curly braces explicitly implementing a function. They are written to compute the value field of the returned `Feature` object. The argument to any `return` statement used can evaluate to anything - even an object - and the resulting value will be converted to a string.

A conjunction is written with the double ampersand operator (`&&`) in between two function expressions. The conjunction of two functions is a function producing a `Feature` whose value is the conjunction of the values of the `Feature`s produced by the conjunction's operands.

Learning function expressions have the following syntax:

```
learn func-exp
using func-exp [, func-exp]*
[from instance-creation-expression]
[with instance-creation-expression]
end
```

The first function expression represents a function that will provide labels for the supervised learning algorithm. The function expressions in the `using` clause do the feature extraction on each object, during both training and evaluation.

The instance creation expression (using the usual Java syntax) in the `from` clause should create an object of a class that implements the `LBJ2.parser.Parser` interface in the library. This clause is optional. If it appears, the LBJ compiler will automatically train the learner represented by this learning function expression at compile-time. Whether it appears or not, the programmer may continue training the learner on-line in the application via methods defined in `LBJ2.learn.Learner` in the library. The `with` clause, which is also optional, specifies the learning algorithm.

A learned function specified in this way is ready to be referenced in other function expressions or, after compilation by the LBJ compiler, to be `import`ed directly into the programmer's application for on-line evaluation.

4.3. Constraints

LBJ constraints are written as arbitrary first order Boolean expressions in terms of functions and the objects in a Java application. The LBJ *constraint statement* syntax is parameterized by Java expressions, so that general constraints may be expressed in terms of data whose exact shape is not known until run-time. The usual operators and quantifiers are provided, as well as the `atleast` and `atmost` quantifiers, which are described below. The only two predicates in the constraint syntax are equality and inequality (meaning string comparison), however their arguments may be arbitrary Java expressions (which will be converted to strings).

Syntactically, an LBJ constraint declaration starts with a header indicating the name of the constraint and the type of object it takes as input, similar to a method declaration with a single parameter. The body of the constraint may then contain arbitrary Java code interspersed with declarative constraint statements. At run-time, when a constraint is invoked, the conjunction of all constraint statements encountered during the invocation is returned as the result.

Each constraint statement contains a single constraint expression which takes one of the following forms:

- An equality predicate ϵ_1 `::` ϵ_2 where ϵ_i is an arbitrary Java expression or the application of an LBJ function to an object.
- An inequality predicate ϵ_1 `!:` ϵ_2 .
- The negation of an LBJ constraint `!`γ .
- The conjunction of two LBJ constraints γ_1 `/\` γ_2 .
- The disjunction of two LBJ constraints γ_1 `\/` γ_2 .
- An implication γ_1 `=>` γ_2 .
- The equivalence of two LBJ constraints γ_1 `<=>` γ_2 .
- A universal quantifier `forall` $(\rho$ `in` $\epsilon)$ γ where ρ is a formal parameter with type τ and identifier ι, ϵ is a Java expression evaluating to a Java `Collection` containing objects of type τ, and γ is an arbitrary constraint expression which may be written in terms of ι.
- An existential quantifier `exists` $(\rho$ `in` $\epsilon)$ γ .
- An "at least" quantifier `atleast` ϵ_1 `of` $(\rho$ `in` $\epsilon_2)$ γ where ϵ_1 is a Java expression evaluating to an `int` and the other parameters play similar roles to those in the universal quantifier.
- An "at most" quantifier `atmost` ϵ_1 `of` $(\rho$ `in` $\epsilon_2)$ γ .
- A constraint invocation `@` $\nu(\epsilon)$ where ν is the constraint's name and ϵ is an arbitrary Java expression.

The `atleast` and `atmost` quantifiers do not add expressivity to FOL. However, an equivalent Boolean expression involving only the conjunction and disjunction operators will contain $\binom{m}{\epsilon_1}$ terms, where m is the size of the `Collection` represented by ϵ_2. Fortunately, as we will see in the next section, these constraints can be translated to a constant number of linear inequalities, which is convenient when solving inference exactly.

4.4. Inference

In LBJ, inference is the glue that combines constraints with learned functions. An LBJ inference is a structure that manages the functions, run-time objects, and constraints. In an LBJ source code, the application of a named inference structure to a learned function involved in the inference produces a new version of that learned function that respects the constraints.

The syntax of an inference starts with a header that indicates the name of the inference and its *head* parameter. (Refer to Figure 2 for an example.) The head parameter (or head object) is an object from which all objects involved in the inference can be reached at run-time. This object need not have the same type as the input parameter of any learned function involved in the inference.

After the header, curly braces surround the body of the inference. The body contains the following four elements. First, it contains a list of "head finder" methods used to locate the head object given an object involved in the inference. Whenever the programmer wishes to use the inference to produce the constrained version of a function involved in the inference, that function's input type must have a head finder method in the inference body. Second, the body specifies how the scores produced by each learned function should be normalized. The LBJ library contains a set of normalizing functions that may be named here. Third, the `subjectto` clause is actually a constraint declaration within which all relevant constraints should be specified or invoked. Finally, the `with` clause specifies which inference algorithm to use.

5. A Solution for Exact Inference

Given an LBJ program $\mathcal{P}$, linear inequalities suitable for use as Integer Linear Programming constraints can be generated as follows. At run-time, the constraint declarations in $\mathcal{C}$, parameterized by Java expressions, become instantiated by Java objects and conjuncted into a single constraint. LBJ's run-time inference mechanism first propositionalizes the instantiated constraint and then recursively generates linear inequalities that hold if and only if the instantiated constraint holds.

5.1. Propositionalization

When propositionalizing an LBJ constraint, there are a few issues to consider. First, we consider the equality and inequality predicates. Their arguments are arbitrary Java expressions and applications of LBJ functions to arbitrary Java expressions. Once instantiated, each argument becomes either a string or an LBJ function application. The comparison of two strings propositionalizes to a constant. The comparison of a function application and a string becomes a Boolean propositional variable. Finally, the comparison of two function applications becomes an expression of the following form: $\bigvee_{s \in \mathbb{S}_1 \cap \mathbb{S}_2} B_{f_1(o_1)=s} \wedge B_{f_2(o_2)=s}$, where $B_{\{\}}$ is a Boolean propositional variable that is true if its subscript is true. Here, the propositionalization process benefits from knowledge of the range of the functions f_1 and f_2; only those strings s shared by their ranges need be included in this expression. This knowledge can always be obtained from the associated scoring functions.

Second, we consider the `atleast` and `atmost` quantifiers. To avoid producing exponentially many literals when propositionalizing these quantifiers, a propositional `atleast` expression is invented. This expression has the form: `atleast` m `of` β, where m is an integer and β is

a list of propositional Boolean expressions. The first order `atleast` expression is then propositionalized directly. The first order `atmost` expression is propositionalized by noticing that `atmost` m `of` $(\rho$ `in` $c)$ $\gamma \equiv$ `atleast` $|c| - m$ `of` $(\rho$ `in` $c)$ `!`γ, where c is a `Collection` and $|c|$ is its size.

Finally, we must consider the form that our final propositionalized constraint will take. Motivated by the algorithm described in Section 5.2, we simplify our propositionalized constraint representation in the following ways. Equivalences and implications are expanded. Conjunctions containing conjunctive terms are recursively flattened into an n-ary conjunction representation. A similar flattening is applied to disjunctions. Negation operators are moved inside of connective expressions using DeMorgan's law until the only negation operators left are the inequalities. Other obvious simplifications, such as $B \wedge$ `true` $\equiv B$ are also applied.

5.2. Generating Linear Inequalities

The algorithm that generates linear equalities suitable for use by an ILP solver is given in Figure 4. It is a recursive algorithm that traverses the syntax tree of a propositionalized constraint expression looking for conjunctions, disjunctions, and `atleast` expressions that contain only equalities and inequalities. Each of these forms of Boolean expression can be translated to a constant number of linear inequalities as follows.

First, consider the form of the simplified, propositional constraint. The top of its syntax tree is a conjunction. Its children are all either literals, disjunctions, or `atleast` expressions. Literals at this top level are translated to linear constraints trivially. Disjunctions of literals at the top level can be translated as a single linear equality of the form: $\sum_i x_i \geq 1$, where each $x_i \in \{0, 1\}$ corresponds to an equality, substituting each inequality with $1 - x_i$. An `atleast` m expression of literals at the top level can be translated to $\sum_i x_i \geq m$, again substituting each inequality with $1 - x_i$. Producing a single constraint for each of these cases works at the top level since an ILP solver will enforce the conjunction of all linear inequalities it is given.

When the children of the top level conjunction do not contain only literals, the algorithm recurses down the syntax tree creating new Boolean variables to represent the more complicated subexpressions. Each new variable $t \in \{0, 1\}$ will come with two linear inequalities that constrain it to take the value 1 if and only if the subexpression it represents is `true`. Then, the parent expression substitutes t for the subexpression, enabling the translation of the parent expression once all its children have been translated.

When a conjunction of literals is found as a subexpression, the new variable t is defined with the following constraints:

```
TranslateToILP(con, t)
  if con is a conjunction
    if t = ∅
      for each child c of con
        TranslateToILP(c, ∅)
    else
      for each child c of con
        TranslateToILP(c,newVariable())
      output equations (3) and (4)
      substitute t for con in con's parent
  else if con is a disjunction
    for each child c of con
      TranslateToILP(c,newVariable())
    if t = ∅, output ∑_i c_i ≥ 1
    else
      output equations (5) and (6)
      substitute t for con in con's parent
  else if con is an atleast m
    for each child c of con
      TranslateToILP(c,newVariable())
    if t = ∅, output ∑_i x_i ≥ m
    else
      output equations (7) and (8)
      substitute t for con in con's parent
  else if con is a literal and t = ∅
    if con is negated, output con = 0
    else, output con = 1
```

Figure 4. Algorithm for generating linear inequality constraints from a propositional logic representation.

$$\sum_i^n x_i - nt \geq 0 \quad (3)$$

$$\sum_i^n x_i - t \leq n - 1 \quad (4)$$

For a disjunction of literals appearing as a subexpression, the new variable t is defined as follows:

$$\sum_i^n x_i - t \geq 0 \quad (5)$$

$$\sum_i^n x_i - nt \leq 0 \quad (6)$$

Finally, for an `atleast` m expression of literals appearing as a subexpression, t is defined as follows:

$$\sum_i^n x_i - mt \geq 0 \quad (7)$$

$$\sum_i^n x_i - nt \leq m - 1 \quad (8)$$

6. Discussion

Learned functions developed with LBJ are truly *on-line*. That is to say that all feature extraction, learning, and inference algorithms necessary to compute the function's final result are performed automatically and on-demand when that function is invoked in the programmer's Java application. This is in stark contrast to the design of most current systems with such components in which each of these three steps is performed separately on all objects before the next step begins. We believe on-line learned functions are much more convenient for the application designer, especially when designing an application that incorporates many learned functions. The main costs incurred by this design are that the memory requirements of all three steps must be supported simultaneously and that the application must be written in Java, which can be less efficient than C or C++.

For example, on a 2.8 Ghz Pentium IV processor, our LBJ POS tagger tags 287,738 words from a corpus of 11997 sentences in 21.8 seconds while consuming about 72 MB of memory. The original C implementation of (essentially) the same tagger takes only 4.4 seconds and consumes about 21 MB of memory. However, the C implementation consists of 1,676 lines of code, and it was necessary to modify external learning algorithm code so that it could be linked into the program. The LBJ POS tagger is implemented in 54 lines of LBJ and 79 lines of Java, requiring no extra provisions.

The chunker we presented in Section 3 was really just a toy example, so comparison against a state of the art implementation isn't really fair. But, for the curious reader, our toy chunker was implemented in 102 lines of LBJ and 431 lines of Java. On the same machine described above, it tagged the same corpus in 7 minutes and 33 seconds using about 131 MB of memory.

Our original state of the art SRL system [8] was implemented in 9,489 lines of C++, PERL, and Bash shell scripts, and involved networking communications with our feature extraction, learning algorithm, and inference codes. The same system was then implemented in 893 lines of LBJ specifying the feature extractors, classifiers, and constraints, and 881 lines of Java for the internal representation and parsing routines. No networking was necessary, and the implementation is dependent only on Java and the LBJ runtime library, making this large system much more portable.

7. Conclusion

We have presented a modeling language for exact, discriminative, global inference. This language provides a convenient syntax for specifying the interactions between functions as arbitrary FOL formulas. We view LBJ as a framework for combining multiple information sources, be they experts, learning algorithms, etc., in a principled way that eases the burden of the programmer. In the future, we plan to study LBJ's scalability as well as exploring alternative inference procedures, and different ways to model them. In particular, we plan to enhance the language with syntactic sugar for designing soft constraints and homegrown inference algorithms.

References

[1] R. Barzilay and M. Lapata. Aggregation via Set Partitioning for Natural Language Generation. In *Proceedings of the Human Language Technology Conference of the NAACL, Main Conference*, pages 359–366, New York City, USA, June 2006. Association for Computational Linguistics.

[2] X. Carreras and L. Màrquez. Introduction to the CoNLL Shared Tasks: Semantic Role Labeling. In *Proceedings of CoNLL-04*, pages 110–113, 2004.

[3] M. Collins. Discriminative training methods for hidden markov models: theory and experiments with perceptron algorithms. In *EMNLP '02: Proceedings of the ACL-02 conference on Empirical methods in natural language processing*, pages 1–8, Morristown, NJ, USA, 2002. Association for Computational Linguistics.

[4] T. Marciniak and M. Strube. Beyond the Pipeline: Discrete Optimization in NLP. In *Proceedings of the Ninth Conference on Computational Natural Language Learning (CoNLL-2005)*, pages 136–143, Ann Arbor, Michigan, June 2005. Association for Computational Linguistics.

[5] B. Milch, B. Marthi, S. Russell, D. Sontag, D. L. Ong, and A. Kolobov. BLOG: Probabilistic models with unknown objects. In *Proc. of the International Joint Conference on Artificial Intelligence*, pages 1352–1359, 2005.

[6] M. Palmer, D. Gildea, and P. Kingsbury. The Proposition Bank: An Annotated Corpus of Semantic Roles. *Computational Linguistics*, 31(1):71–106, March 2005.

[7] A. Pfeffer. IBAL: A probabilistic rational programming language. In *IJCAI*, pages 733–740, 2001.

[8] V. Punyakanok, D. Roth, and W. Yih. The Necessity of Syntactic Parsing for Semantic Role Labeling. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1117–1123, 2005.

[9] V. Punyakanok, D. Roth, W. Yih, and D. Zimak. Semantic Role Labeling via Integer Linear Programming Inference. In *Proc. the International Conference on Computational Linguistics (COLING)*, pages 1346–1352, Geneva, Switzerland, August 2004.

[10] M. Richardson and P. Domingos. Markov logic networks. *Mach. Learn.*, 62(1-2):107–136, 2006.

[11] D. Roth and W. Yih. Integer linear programming inference for conditional random fields. In *Proc. of the International Conference on Machine Learning*, pages 737–744, 2005.

[12] T. Sato and Y. Kameya. PRISM: A symbolic-statistical modeling language. In *Proc. of the International Joint Conference on Artificial Intelligence*, pages 1330–1335, 1997.

[13] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, San Francisco, California, 2nd edition, 2005.