An Introduction to

Learning Based Java

Version 2.8.2

Nicholas Delmonico Rizzolo

April 1, 2013

Contents

Chapter 1

Introduction

Learning Based Java is a modeling language for the rapid development of software systems with one or more learned functions, designed for use with the Java programming language. LBJ offers a convenient, declarative syntax for classifier and constraint definition directly in terms of the objects in the programmer's application. With LBJ, the details of feature extraction, learning, model evaluation, and inference are all abstracted away from the programmer, leaving him to reason more directly about his application.

1.1 Motivation

Many software systems are in need of functions that are simple to describe but that no one knows how to implement. Recently, more and more designers of such systems have turned to machine learning to plug these gaps. Given data, a discriminative machine learning algorithm yields a function that classifies instances from some problem domain into one of a set of categories. For example, given an instance from the domain of email messages (i.e., given an email), we may desire a function that classifies that email as either "spam" or "not spam". Given data (in particular, a set of emails for which the correct classification is known), a machine learning algorithm can provide such a function. We call systems that utilize machine learning technology learning based programs.

Modern learning based programs often involve several learning components (or, at least a single learning component applied repeatedly) whose classifications are dependent on each other. There are many approaches to designing such programs; here, we focus on the following approach. Given data, the various learning components are trained entirely independently of each other, each optimizing its own loss function. Then, when the learned functions are applied in the wild, the independent predictions made by each function are reconciled according to user specified constraints. This approach has been applied successfully to complicated domains such as Semantic Role Labeling (Punyakanok, Roth, & Yih, 2008).

1.2 LBJ

Learning Based Java (LBJ) is a modeling language that expedites the development of learning based programs, designed for use with the JavaTM programming language. The LBJ compiler accepts the programmer's classifier and constraint specifications as input, automatically generating efficient

Java code and applying learning algorithms (i.e., performing training) as necessary to implement the classifiers' entire computation from raw data (i.e., text, images, etc.) to output decision (i.e., part of speech tag, type of recognized object, etc.). The details of feature extraction, learning, model evaluation, and inference (i.e., reconciling the predictions in terms of the constraints at runtime) are abstracted away from the programmer.

Under the LBJ programming philosophy, the designer of a learning based program will first design an object-oriented internal representation (IR) of the application's raw data using pure Java. For example, if we wish to write software dealing with emails, then we may wish to define a Java class named Email. An LBJ source file then allows the programmer to define classifiers that take Emails as input. A classifier is merely any method that produces one or more discrete or real valued classifications when given a single object from the programmer's IR. It might be hard-coded using explicit Java code (usually for use as a feature extractor), or learned from data (e.g., labeled example Emails) using other classifiers as feature extractors.

Feature extraction and learning typically produce several different intermediate representations of the data they process. The LBJ compiler automates these processes, managing all of their intermediate representations automatically. An LBJ source file also acts as a Makefile of sorts. When you make a change to your LBJ source file, LBJ knows which operations need to be repeated. For example, when you change the code in a hard-coded classifier, only those learned classifiers that use it as a feature will be retrained. When you change only a learning algorithm parameter, LBJ skips feature extraction and goes straight to learning.

LBJ is supported by a library of interfaces and classes that implement a standardized functionality for features and classifiers. The library includes learning and inference algorithm implementations, general purpose and domain specific internal representations, and domain specific parsers.

Chapter 2

LBJ Definitions

The terms defined below are used throughout this manual. Their definitions have been carefully formulated to facilitate the design of a modeling language that uses them as building blocks. They are intended to disambiguate among the terms' various usages in the literature, and in doing so they encapsulate the LBJ programming philosophy. For those experienced with ML and programming with learning algorithms, the biggest differences from a more typical ML programming philosophy are:

- The word "classifier" does not imply learning. Naïve Bayes, SVM, Perceptron, etc. are *not* classifiers; they are learning algorithms.
- Classifiers (learned and otherwise) should work directly with (internal representations of) raw data (e.g. text, images, etc.), and they should return values that are directly useful to the application.
- Any interaction among classifiers' computations should be completely transparent to the programmer. In particular, when classifiers' outputs are constrained with respect to each other, the classifiers should automatically return values that respect the constraints.

Feature

A feature is a data type which has both a name and a value. There are two types of features in LBJ; discrete and real. The name of the feature in either case is always a `String`. A discrete feature has a value of type `String` assumed to come from some finite set of unordered values associated with the feature's name (although it is not necessary to know what values that set contains ahead of time). A real feature has a value of type `double`.

Most learning algorithms use features to index the parameters whose values are determined during training.[1] In the case of real features, the name of the feature alone identifies the corresponding parameter. In the case of discrete features, the corresponding parameter is identified by the name and value together.[2] The only exception to this rule is when a discrete feature is known

[1] An exception to this rule would be a decision tree learning algorithm, which doesn't really have parameters, per se.

[2] Note again that a decision tree learning algorithm would not need to "split up" the values of a discrete feature so that they represent separate features in this way. It would simply use a single branching point with many branches. Anyway, it's a mute point, since LBJ does not currenlty provide any decision tree learning algorithm implementations.

to allow only two different values. Such a feature is equivalent to a real feature that can take only the values 0 and 1; in particular, its name alone identifies the corresponding learned parameter.

Classifier

A classifier is a method that takes exactly one object from the application domain's internal representation as input and produces zero or more features as output. When defining a classifier, LBJ's syntax requires the programmer to specify the classifier's input type, what type of feature(s) are produced, and whether or not multiple features may be produced. Depending on how the classifier is declared, the programmer can be given a greater degree of control over the features:

- A classifier may be defined to produce exactly one feature. In this case, the name of the feature produced is taken to be the name of the classifier, leaving only the value to be computed by the classifier.

- A classifier may also be defined as a *feature generator*. In this case, both the names and the values of the produced features are computed by the classifier. The name of the classifier will also be stored inside the feature to disambiguate with similar features produced by other classifiers.

A classifier may be coded explicitly using arbitrary Java, it may be composed from other classifiers using LBJ's classifier operators, or it may be learned from data as a function of the features produced by other classifiers with a special syntax discussed later.

Learner

A learner or learning classifier is a classifier capable of changing its implementation with experience. The methods used for effecting that change are commonly referred to as the *learning algorithm*. In general, the programmer is encouraged to simply call on a learning algorithm implemented in LBJ's library, but it is also possible to implement a new learning algorithm in a separate Java source code and to call it from LBJ source code in the same way.

Examples and Example Objects

An example object is simply an instance from the internal representation of the raw data. It can be any object whatsoever. Examples are collections of features extracted from a given example object using classifiers. They are taken as input by a learning algorithm both at training time and at testing time. Thus, as we will see, every learner starts with an example object, applies a set of classifiers to that object to create an example, and then sends that example to its learning algorithm for processing.

An example may or may not come with a *label*, which is just a feature that indicates the correct classification of the corresponding example object. Labels are extracted by classifiers, just like any other feature. The only difference is that a classifier designated as a label extractor will only be called by a learner during training.

Parser

A parser is a function that takes raw data as input and instantiates example objects as output. If a learner is associated with a parser inside an LBJ source code, the LBJ will train that learner using the example objects produced by the parser at LBJ-compile time. There are several domain specific parsers defined in the LBJ library, but the programmer will often need to implement his own in a separate Java source file.

Inference

Inference is the process through which discrete classifications made about objects are reconciled with global constraints over those classifications. Constraints are written by the programmer using first order logic and may involve any number of classifiers and objects. LBJ automatically translates those constraints to linear inequalities for use in an Integer Linear Program at run-time.

Application

The application is the data processing code written in pure Java that works directly with the (internal representation of) the raw data and has need to classify elements of that data. Because it calls classifiers defined only in the LBJ source code, it typically cannot be compiled until after LBJ has compiled the LBJ source file.

Chapter 3

Tutorial: 20 Newsgroups

We begin our discussion of the LBJ language with a tutorial that illustrates its most common usage. This tutorial is intended for a first time user of the language. It introduces the syntax for both single feature and feature generating hard-coded classifiers, as well as the syntax for declaring a learner. Next, it shows how the learner (or any other classifier) declared in the LBJ source code can be imported and used in the Java application. Finally, it discusses how to use the LBJ compiler on the command line to fully compile our learning based program. Throughout the tutorial, we'll be using the famous 20 Newsgroups corpus[1] as our training data.

3.1 Setting Up

Suppose we want to classify newsgroup posts according to the newsgroup to which each post is best suited. Such a classifier could be used in a newsgroup client application to automatically suggest an appropriate destination for a new post. It is plausible that these classifications could be made as a function of the words that appear in them. For example, the word "motor" is likely to appear more often in `rec.autos` or `rec.motorcycles` than in `alt.atheism`. However, we do not want to manually invent these associations one at a time, so we turn to LBJ.

To use LBJ, we first need to decide on an object oriented internal representation. In this case, it makes sense to define a class named `Post` that stores the contents of a newsgroup post. Figure 3.1 shows a skeleton for such a class. There, we see that space has been allocated for several fields that we might expect in a newsgroup post, namely the "From" and "Subject" fields from the header and the body of the post. We have chosen to represent the body as a two dimensional array; one dimension for the lines in the body, and the other for the words in each line.

Finally, we have a field called `newsgroup`. It may seem counterintuitive to include a field to store this information since it is exactly the classification we aim to compute. LBJ's supervised learning algorithms will need this information, however, since it labels the example object. Furthermore, at test time, our newsgroup client application may fill this field with the newsgroup in which the post was encountered or in which the user intends to post it, and the learned classifier will simply ignore it at that point.

We'll also need to implement a parser that knows how to create `Post` objects when given the raw data in a file or files. In LBJ, a parser is any class that implements the `LBJ2.parse.Parser` interface. This is a simple interface that requires only three methods be defined. First, the `next()`

[1] `http://people.csail.mit.edu/jrennie/20Newsgroups`

```
1.  public class Post {
2.    private String newsgroup; ⟪The label of the post.⟫
3.    private String fromHeader;
4.    private String subjectHeader;
5.    private String[][] body;
6.    ⟪Accessor methods omitted for brevity.⟫
7.  }
```

Figure 3.1. Class `Post` represents a newsgroup post.

```
1.  import LBJ2.parse.LineByLine;
2.
3.  public class NewsgroupParser extends LineByLine {
4.    public NewsgroupParser(String file) { super(file); }
5.    public Object next() {
6.      String file = readLine();
7.      if (file == null) return null; ⟪No more examples.⟫
8.      return new Post(file);
9.    }
10. }
```

Figure 3.2. Class `NewsgroupParser` instantiates `Post` objects and returns them one at a time via the `next()` method.

method takes no arguments and returns a single example `Object` (of any type in general, but in this case, it will be a `Post`). The LBJ compiler will call this method repeatedly to retrieve training example objects until it returns `null`. Next, the `reset()` method rewinds the parser back to the beginning of the raw data input it has been reading. Finally, the `close()` method closes any streams the parser may have open and frees any other system resources it may be using.

The LBJ library comes with several parsers that read plain text. While it does not include a parser for newsgroup posts, we can still make use of `LBJ2.parse.LineByLine`, which will at least take care of the boilerplate code necessary to read text out of a file. This abstract class also provides implementations of the `reset()` and `close()` methods. The `NewsgroupParser` class in Figure 3.2 simply extends it to take advantage of that functionality; it won't be necessary to override `reset()` or `close()`. `NewsgroupParser` takes as input a file containing the names of other files, assuming that each of those files represents a single newgroup post. For brevity, we have hidden in `Post`'s constructor the code that actually does the work of filling the fields of a `Post` object.

With `Post` and `NewsgroupParser` ready to go, we can now define in the LBJ source code a hard-coded classifier that identifies which words appear in each post and a learning classifier that categorizes each post based on those words.

```
1.  discrete% BagOfWords(Post post) <- {
2.    for (int i = 0; i < post.bodySize(); ++i)
3.      for (int j = 0; j < post.lineSize(i); ++j) {
4.        String word = post.getBodyWord(i, j);
5.        if (word.length() > 0 && word.substring(0, 1).matches("[A-Za-z]"))
6.          sense word;
7.      }
8.  }
9.
10. discrete NewsgroupLabel(Post post) <- { return post.getNewsgroup(); }
11.
12. discrete NewsgroupClassifier(Post post) <-
13. learn NewsgroupLabel
14.   using BagOfWords
15.   from new NewsgroupParser("data/20news.train.shuffled") 40 rounds
16.   with SparseNetworkLearner {
17.     SparseAveragedPerceptron.Parameters p =
18.         new SparseAveragedPerceptron.Parameters();
19.     p.learningRate = .1;
20.     p.thickness = 3;
21.     baseLTU = new SparseAveragedPerceptron(p);
22.   }
23. end
```

Figure 3.3. A simple, learned newsgroup classifier.

3.2 Classifier Declarations

Given the internal representation developed in the previous section, the LBJ code in Figure 3.3 can be used to train a learned newsgroup classifier. It involves a single feature extraction classifier named `BagOfWords`, a label classifier named `NewsgroupLabel` to provide labels during training, and a multi-class classifier named `NewsgroupClassifier` that predicts a newsgroup label. It also assumes that the `Post` class and the parser `NewsgroupParser` (or their source files) are available on the `CLASSPATH`. To see the code in action, download the source distribution[2] from our website – it includes the data and all the classes mentioned above – and run `./train.sh` (assuming that LBJ is already on your `CLASSPATH`). We'll now take a closer look at how it works.

3.2.1 Hard-coded classifiers

An LBJ source file is a list of declarations. The simplest in Figure 3.3 is contained entirely on line 10. It consists of the classifier's *signature* and a hard-coded *classifier expression* separated by a left arrow indicating assignment. In the classifier's signature, we see its return type (a single discrete feature) as well as its input type (an object of type `Post`). All LBJ classifiers take a single object (of any type) as input. It is up to the programmer to ensure that all information pertinent to the classifiers is accessible from that object. The return type, however, is not quite so restrictive.

[2] `http://cogcomp.cs.illinois.edu/software/20news.tgz`

Returned features may be either `discrete` or `real`, and a classifier may return either a single feature (as on line 10) or multiple features (as indicated on line 1 with the `\%` symbol). When a classifier can return multiple features, we call it a *feature generator*.

On the right hand side of the left arrow is placed a classifier expression. There are many types of classifier expression, and the two most common are on display in this figure. `BagOfWords` and `NewsgroupLabel` are defined with *hard-coded classifier expressions*, while `NewsgroupClassifier` is defined with a *learning classifier expression*. When hard-coding the behavior of a classifier, the programmer has Java 1.4 syntax at his disposal to aid in computing his features' values, plus some additional syntactic sugar to make that type of computation easier. For example, the `sense` statement on line 6 creates a feature which will eventually be returned, but execution of the method continues so that multiple features can be "sensed." Note that only feature generators can use the `sense` statement, and only classifiers returning a single feature can use Java's `return` statement (as on line 10).

After everything is said and done, we end up with two hard-coded classifiers. One is a simple, one feature classifier that merely returns the value of the `Post.newsgroup` field (via the `getNewsgroup()` method, since `Post.newsgroup` is private). The other loops over all the words in the post returning each as a separate feature.

3.2.2 Learners

`NewsgroupClassifier` on line 12 of Figure 3.3 is not specified in the usual, procedural way, but instead as the output of a learning algorithm applied to data. The verbose learning classifier expression syntax says that this classifier will `learn` to mimic an oracle (line 13), `using` some feature extraction classifiers (line 14), `from` some example objects (line 15), `with` a learning algorithm (lines 16 through 22). The expression ends with the `end` keyword (line 23). In this case, the oracle is `NewsgroupLabel`, the only feature extraction classifier is `BagOfWords`, the example objects come from `NewsgroupParser`, and the learning algorithm is `SparseNetworkLearner`. We explore each of these ideas in more detail below.

`learn`

> We say that `NewsgroupClassifier` is trying to mimic `NewsgroupLabel` because it will attempt to return features with the same values and for the same example objects that `NewsgroupLabel` would have returned them. Note that the particular feature values being returned have not been mentioned; they are induced by the learning algorithm from the data. We need only make sure that the return type of the label classifier is appropriate for the selected learning algorithm.

`using`

> The argument to the `using` clause is a single classifier expression. As we can see from this example code, the name of a classifier qualifies. The only restriction is that this classifier expression must have an input type that allows it to take instances of `NewsgroupClassifier`'s input type. LBJ also provides a comma operator for constructing a feature generator that simply returns all the features returned by the classifiers on either side of the comma. This way, we can include as many features as we want simply by listing classifiers separated by commas.

`from`

The `from` clause supplies a data source by instantiating a parser. The objects returned by this parser's `next()` method must be instances of `NewsgroupClassifier`'s input type. LBJ can then extract features via the using clause and train with the learning algorithm. This clause also gives the programmer the opportunity to iterate over the training data if he so desires. The optional `rounds` clause is part of the `from` clause, and it specifies how many times to iterate.

`with`

The argument to the `with` clause names a learning algorithm (any class extending `LBJ2.learn.Learner` accessible on the `CLASSPATH`) and allows the programmer to set its parameters. For example, `learningRate` (line 19) and `thickness` (line 20) are parameters of the `SparseAveragedPerceptron` learning algorithm, while `baseLTU` (line 21) is a parameter of the `SparseNetworkLearner` learning algorithm.

From these elements, the LBJ compiler generates Java source code that performs feature extraction, applies that code on the example objects to create training examples, and trains our learner with them. The resulting learner is, in essence, a Java method that takes an example object as input and returns the predicted newsgroup in a string as output. Note that the code does not specify the possible newsgroup names or any other particulars about the content of our example objects. The only reason that this LBJ code results in a newsgroup classifier is because we give it training data that induces one. If we want a spam detector instead, we need only change data sources; the LBJ code need not change.[3]

3.3 Using `NewsgroupClassifier` in a Java Program

Now that we've specified a learned classifier, the next step is to write a pure Java application that will use it once it's been trained. This section first introduces the methods every automatically generated LBJ classifier makes available within pure Java code. These methods comprise a simple interface for predicting, online learning, and testing with a classifier.

3.3.1 Getting Started

We assume here that all learning will take place during the LBJ compilation phase, which we'll discuss in Section 3.4. (It is also possible to learn online, i.e. while the application is running, which we'll discuss in Section 3.3.3.) To gain access to the learned classifier within your Java program, simply instantiate an object of the classifier's generated class, which has the same name as the classifier.

```
NewsgroupClassifier ngClassifier = new NewsgroupClassifier();
```

The classifier is now ready to make predictions on example objects. `NewsgroupClassifier` was defined to take `Post` objects as input and to make a discrete prediction as output. Thus, if we have a `Post` object available, we can retrieve `NewsgroupClassifier`'s prediction like this:

[3] Of course, we may want to change the names of our classifiers in that case for clarity's sake.

```
Post post = ...
String prediction = ngClassifier.discreteValue(post);
```

The prediction made by the classifier will be one of the string labels it observed during training. And that's it! The programmer is now free to use the classifier's predictions however s/he chooses.

There's one important technical point to be aware of here. The instance we just created of class `NewsgroupClassifier` above does not actually contain the model that LBJ learned for us. It is merely a "clone" object that contains internally a reference to the real classifier. Thus, if our Java application creates instances of this class in different places and performs any operation that modifies the behavior of the classifier (like online learning), all instances will appear to be affected by the changes. For simple use cases, this will not be an issue, but see Section 3.3.4 for details on gaining direct access to the model.

3.3.2 Prediction Confidence

We've already seen how to get the prediction from a discrete valued classifier. This technique will work no matter how the classifier was defined; be it hard-coded, learned, or what have you. When the classifier is learned, it can go further than merely providing the prediction value it likes the best. In addition, it can provide a score for every possible prediction it chose amongst, thereby giving an indication of how confident the classifier is in its prediction. The prediction with the highest score is the one selected.

Scores are returned by a classifier in a `ScoreSet` object by calling the `score(Object)` method, passing in the same example object that you would have passed to `discreteValue(Object)`. Once you have a `ScoreSet` you can get the score for any particular prediction value using the `get(String)` method, which returns a `double`. Alternatively, you can retrieve all scores in an array and iterate over them, like this:

```
ScoreSet scores = ngClassifier.scores(post);
Score[] scoresArray = scores.toArray();
for (Score score : scoresArray)
  System.out.println("prediction: " + score.value
                     + ", score: " + score.score);
```

Finally, LBJ also lets you define real valued classifiers which return doubles in the Java application. If you have such a classifier, you can retreive its prediction on an example object by calling the realValue(Object) method:

```
double prediction = realClassifier.realValue(someExampleObject);
```

3.3.3 Learning

As mentioned above, most classifiers are learned during the LBJ phase of compilation (see Section 3.4 below). In addition, a classifier generated by the LBJ compiler can also continue learning from labeled examples in the Java application. Since `NewsgroupClassifier` takes a `Post` object as input, we merely have to get our hands on such an object, stick the label in the `newsgroup` field (since that's where the `NewsgroupLabel` classifier will look for it), and pass it to the classifier's `learn(Object)` method.

Now that we know how to get our classifier to learn, let's see how to make it forget. The contents of a classifier can be completely cleared out by calling the `forget()` method. After this method is called, the classifier returns to the state it was in before it observed any training examples. One reason to forget everything a classifier has learned is to try new learning algorithm parameters (e.g. learning rates, thresholds, etc.). All LBJ learning algorithms provide an inner class named `Parameters` that contains default settings for all their parameters. Simply instantiate such an object, overwrite the parameters that need to be updated, and call the `setParameters(Parameters)` method. For example:

```
ngClassifier.forget();
SparseAveragedPerceptron.Parameters ltuParameters =
    new SparseAveragedPerceptron.Parameters();
ltuParameters.thickness = 12;
NewsgroupClassifier.Parameters parameters =
    new NewsgroupClassifier.Parameters();
parameters.baseLTU = new SparseAveragedPerceptron(ltuParameters);
ngClassifier.setParameters(parameters);
```

This particular example is complicated by the fact that our newsgroup classifier is learned using `SparseNetworkLearner`, an algorithm that uses another learning algorithm with its own parameters as a subroutine. But the technique is the same. At this point, the classifier is re-initialized with a new `thickness` setting and is ready for new training examples.

3.3.4 Saving Your Work

If we've done any `forget()`ing and/or `learn()`ing within our Java application, we'll probably be interested in saving what we learned at some point. No problem; simply call the `save()` method.

```
classifier.save();
```

This operation overwrites the *model* and *lexicon* files that were originally generated by the LBJ compiler. A *model file* stores the values of the learned parameters (not to be confused with the manually set learning algorithm parameters mentioned above). A *lexicon file* stores the classifier's feature index, used for quick access to the learnable parameters when training for multiple rounds. These files are written by the LBJ compiler and by the `save()` method (though

only initially; see below) in the same directory where the `NewsgroupClassifier.class` file is written.

We may also wish to train several versions of our classifier; perhaps each version will use different manually set parameters. But how can we do this if each instance of our `NewsgroupClassifier` class is actually a "clone", merely pointing to the real classifier object? Easy: just use the `NewsgroupClassifier` constructor that takes model and lexicon filenames as input:

```
NewsgroupClassifier c2 = new NewsgroupClassifier( "myModel.lc", "myLexicon.lex");
```

This instance of our classifier is not a clone, simply by virtue of our chosen constructor. It has its own completely independent learnable parameters. Furthermore, if `myModel.lc` and `myLexicon.lex` exist, they will be read from disk into `c2`. If not, then calling this constructor creates them. Either way, we can now train our classifier however we choose and then simply call `c2.save()` to save everything into those files.

3.4 Compiling Our Learning Based Program with LBJ

Referring once again to this newsgroup classifier's source distribution[4], we first examine our chosen directory structure starting from the root directory of the distribution.

```
$ ls
20news.lbj class      lbj        test.sh
README     data       src        train.sh
$ ls src/dssi/news
NewsgroupParser.java     NewsgroupPrediction.java Post.java
```

We see there is an LBJ source file `20news.lbj` in the root directory, and in `src/dssi/news` we find plain Java source files implementing our internal representation (`Post.java`), a parser that instantiates our internal representation (`NewsgroupParser.java`), and a program intended use our trained classifier to make predictions about newsgroups (`NewsgroupPrediction.java`). Note that the LBJ source file and all these plain Java source files declare `package dssi.news;`. The root directory also contains two directories `class` and `lbj` which are initially empty. They will be used to store all compiled Java class files and all Java source files *generated by the LBJ compiler* respectively. Keeping all these files in separate directories is not a requirement, but many developers find it useful to reduce clutter around the source files they are editing.

[4] `http://cogcomp.cs.illinois.edu/software/20news.tgz`

To compile the LBJ source file using all these directories as intended, we run the following command:

```
$ java -Xmx512m -cp $CLASSPATH:class LBJ2.Main \
       -sourcepath src \
       -gsp lbj \
       -d class \
       20news.lbj
```

This command runs the LBJ compiler on `20news.lbj`, generating a new Java source file for each of the classifiers declared therein. Since `20news.lbj` mentions both the `Post` and `NewsgroupParser` classes, their definitions (either compiled class files or their original source files) must be available within a directory structure that mirrors their package names. We have provided their source files using the `-sourcepath src` command line flag. The `-gsp lbj` (generated source path) flag tells LBJ to put the new Java source files it generates in the `lbj` directory, and the `-d class` flag tells LBJ to put class files in the `class` directory. For more information on the LBJ compiler's command line usage, see Chapter 6.

But the command does more than that; it also trains any learning classifiers on the specified training data, so that the compiled class files are ready to be used in new Java programs just like any other class can be. The fact that their implementations came from data is immaterial; the new Java program that uses these learned classifiers is agnostic to whether the functions it is calling are learned or hard-coded. Its output will look like this:

```
Generating code for BagOfWords
Generating code for NewsgroupLabel
Generating code for NewsgroupClassifier
Compiling generated code
Training NewsgroupClassifier
  NewsgroupClassifier, pre-extract: 0 examples at Sun Mar 31 10:48...
  NewsgroupClassifier, pre-extract: 16828 examples at Sun Mar 31 10:49...
  NewsgroupClassifier: Round 1, 0 examples processed at Sun Mar 31 10:49...
  NewsgroupClassifier: Round 1, 16828 examples processed at Sun Mar 31 10:49...
  NewsgroupClassifier: Round 2, 0 examples processed at Sun Mar 31 10:49...
  ...
Writing NewsgroupClassifier
Compiling generated code
```

The compiler tells us which classifiers it is generating code for and which it is training. Because we have specified `progressOutput 20000` in `NewsgroupClassifier`'s specification (see the distribution's `20news.lbj` file), we also get messages updating us on the progress being made during training. We can see here that the first stage of training is a "pre-extraction" stage in which a feature index is compiled, and all `Post` objects in our training set are converted to feature vectors based on the index. Then the classifier is trained over those vectors for 40 rounds. The entire process should take under 2 minutes on a modern machine.

If you're curious, you can also look at the files that have been generated:

```
$ ls lbj/dssi/news
BagOfWords.java          NewsgroupClassifier.java
NewsgroupClassifier.ex   NewsgroupLabel.java
$ ls class/dssi/news
BagOfWords.class                      NewsgroupLabel.class
NewsgroupClassifier$Parameters.class NewsgroupParser.class
NewsgroupClassifier.class             Post$1.class
NewsgroupClassifier.lc                Post.class
NewsgroupClassifier.lex
```

The `lbj` directory now contains a `dssi/news` subdirectory containing our classifier's Java implementations, as well as the pre-extracted feature vectors in the `NewsgroupClassifier.ex` file. In the `class/dssi/news` directory, we find the class files compiled from all our hard-coded and generated Java source files, as well as `NewsgroupClassifier.lc` and `NewsgroupClassifier.lex`, which contain `NewsgroupClassifier`'s learned parameters and its feature index (a.k.a. *lexicon*) respectively.

Finally, it's time to compile `NewsgroupPrediction.java`, the program that calls our learned classifier to make predictions about new posts.

```
$ javac -cp $CLASSPATH:class \
        -sourcepath src \
        -d class \
        src/dssi/news/NewsgroupPrediction.java
```

Notice that the command line flags we gave to the LBJ compiler previously are very similar to those we give the Java compiler now. We can test out our new program like this:

```
$ java -Xmx512m -cp $CLASSPATH:class dssi.news.NewsgroupPrediction \
       $(head data/20news.test.shuffled)
data/alt.atheism/53531: alt.atheism
data/talk.politics.mideast/76075: talk.politics.mideast
data/sci.med/59050: sci.med
data/rec.sport.baseball/104591: rec.sport.baseball
data/comp.windows.x/67088: comp.windows.x
data/rec.motorcycles/103131: rec.autos
data/sci.crypt/15215: sci.crypt
data/talk.religion.misc/84195: talk.religion.misc
data/sci.electronics/54094: sci.electronics
data/comp.os.ms-windows.misc/10793: comp.os.ms-windows.misc
```

Post `rec.motorcycles/103131` was misclassified as `rec.autos`, but other than that, things are going well.

3.5 Testing a Discrete Classifier

When a learned classifier returns discrete values, LBJ provides the handy `TestDiscrete`[5] class for measuring the classifier's prediction performance. This class can be used either as a stand-alone program or as a library for use inside a Java application. In either case, we'll need to provide `TestDiscrete` with the following three items:

- The classifier whose performance we're measuring (e.g. `NewsgroupClassifier`).
- An *oracle* classifier that knows the true labels (e.g. `NewsgroupLabel`).
- A *parser* (i.e., any class implementing the `Parser` interface) that returns objects of our classifiers' input type.

3.5.1 On the Command Line

If we'd like to use `TestDiscrete` on the command line, the parser must provide a constructor that takes a single `String` argument (which could be, e.g., a file name) as input. `NewsgroupClassifier` uses the `NewsgroupParser` parser, which meets this requirement, so we can test our classifier on the command line like this:

```
$ java -Xmx512m -cp $CLASSPATH:class LBJ2.classify.TestDiscrete \
    dssi.news.NewsgroupClassifier \
    dssi.news.NewsgroupLabel \
    dssi.news.NewsgroupParser \
    data/20news.test
```

The output of this program is a table of the classifier's performance statistics broken down by label. For a given label l, the statistics are based on the quantity of examples with that gold truth label c_l, the quantity of examples predicted to have that label by the classifier $\bar{c}_l$, and the overlap of these two sets, denoted $c_l \wedge \bar{c}_l$ (i.e., the quantity of examples correctly predicted to have that label). Based on these definitions, the table has the following columns:

1. the label l,
2. the classifier's precision on l, $p_l = \frac{c_l \wedge \bar{c}_l}{\bar{c}_l} \times 100\%$
3. the classifier's recall on l, $r_l = \frac{c_l \wedge \bar{c}_l}{c_l} \times 100\%$
4. the classifier's F_1 on l, $F_1(l) = 2\frac{p_l r_l}{p_l + r_l} \times 100\%$
5. the label count c_l, and
6. the prediction count $\bar{c}_l$.

At the bottom of the table will always be the overall accuracy of the classifier. For the `NewsgroupClassifier`, we get this output:

[5] `http://cogcomp.cs.uiuc.edu/software/doc/LBJ2/library/LBJ2/classify/TestDiscrete.html`

```
         Label           Precision Recall   F1   LCount PCount
-------------------------------------------------------------
alt.atheism                 80.000 80.000 80.000     80     80
comp.graphics               78.814 77.500 78.151    120    118
comp.os.ms-windows.misc     80.198 79.412 79.803    102    101
comp.sys.ibm.pc.hardware    74.074 79.208 76.555    101    108
comp.sys.mac.hardware       80.000 77.551 78.756     98     95
comp.windows.x              82.955 85.882 84.393     85     88
misc.forsale                70.588 80.769 75.336    104    119
rec.autos                   77.551 89.063 82.909    128    147
rec.motorcycles             78.571 84.615 81.481    104    112
rec.sport.baseball          81.197 91.346 85.973    104    117
rec.sport.hockey            90.291 90.291 90.291    103    103
sci.crypt                   90.816 85.577 88.119    104     98
sci.electronics             77.570 85.567 81.373     97    107
sci.med                     83.019 88.000 85.437    100    106
sci.space                   91.837 78.947 84.906    114     98
soc.religion.christian      84.946 79.000 81.865    100     93
talk.politics.guns          86.747 72.727 79.121     99     83
talk.politics.mideast       91.262 89.524 90.385    105    103
talk.politics.misc          85.915 76.250 80.795     80     71
talk.religion.misc          86.792 63.889 73.600     72     53
-------------------------------------------------------------
Accuracy                    82.150   -      -      -      2000
```

The `TestDiscrete` class also supports the notion of a *null label*, which is a label intended to represent the absense of a prediction. The 20 Newsgroups task doesn't make use of this concept, but if our task were, e.g., named entity classification in which every phrase is potentially a named entity, then the classifier will likely output a prediction we interpret as meaning "this phrase is not a named entity." In that case, we will also be interested in overall precision, recall, and F_1 scores aggregated over the non-null labels. On the `TestDiscrete` command line, all arguments after the four we've already seen are optional null labels. The output with a single null label "`O`" might look like this (note the `Overall` row at the bottom):

```
 Label   Precision Recall   F1   LCount PCount
-----------------------------------------------
LOC         88.453 87.153 87.798   1837   1810
MISC        83.601 79.067 81.271    922    872
ORG         76.226 76.510 76.368   1341   1346
PER         86.554 88.762 87.644   1842   1889
-----------------------------------------------
O            0.000  0.000  0.000    581    606
-----------------------------------------------
Overall     84.350 83.995 84.172   5942   5917
Accuracy    76.514                       6523
```

3.5.2 In a Java Program

Alternatively, we can call `TestDiscrete` from within our Java application. This comes in handy if our parser's constructor isn't so simple, or when we'd like to do further processing with the performance numbers themselves. The simplest way to do so is to pass instances of our classifier, labeler, and parser to `TestDiscrete`, like this:

```
NewsgroupLabel oracle = new NewsgroupLabel();
Parser parser = new NewsgroupParser("data/20news.test");
TestDiscrete tester = TestDiscrete.testDiscrete(classifier, oracle, parser);
tester.printPerformance(System.out);
```

This Java code does exactly the same thing as the command line above. We can also exert more fine grained control over the computed statistics. Starting from a new instance of `TestDiscrete`, we can call `reportPrediction(String,String)` every time we acquire both a prediction value and a label. Then we can either call the `printPerformance(PrintStream)` method to produce the standard output in table form or any of the methods whose names start with `get` to retrieve individual statistics. The example code below retrieves the overall precision, recall, F_1, and accuracy measures in an array.

```
TestDiscrete tester = new TestDiscrete();
...
tester.reportPrediction(classifier.discreteValue(ngPost),
                        oracle.discreteValue(ngPost));
...
double[] performance = tester.getOverallStats();
System.out.println("Overall Accuracy: " + performance[3]);
```

Chapter 4

The LBJ Language

Now that we have defined the building blocks of classifier computation, we next describe LBJ's syntax and semantics for programming with these building blocks.

Like a Java source file, an LBJ source file begins with an optional package declaration and an optional list of import declarations. Next follow the definitions of classifiers, constraints, and inferences. Each will be translated by the LBJ compiler into a Java class of the same name. If the package declaration is present, those Java classes will all become members of that package. Import declarations perform the same function in an LBJ source file as in a Java source file.

4.1 Classifiers

In LBJ, a classifier can be defined with Java code or composed from the definitions of other classifiers using special operators. As such, the syntax of classifier specification allows the programmer to treat classifiers as expressions and assign them to names. This section defines the syntax of classifier specification more precisely, including the syntax of classifiers learned from data. It also details the behavior of the LBJ compiler when classifiers are specified in terms of training data and when changes are made to an LBJ source file.

4.1.1 Classifier Declarations

Classifier declarations are used to name classifier expressions (discussed in Section 4.1.2). The syntax of a classifier declaration has the following form:

feature-type name`(`*type name*`)`
  [`cached` | `cachedin` *field-access*] `<-`
  classifier-expression

A classifier declaration names a classifier and specifies its input and output types in its header, which is similar to a Java method header. It ends with a left arrow indicating assignment and a classifier expression which is assigned to the named classifier.

The optional `cached` and `cachedin` keywords are used to indicate that the result of this classifier's computation will be cached in association with the input object. The `cachedin` keyword

instructs the classifier to cache its output in the specified field of the input object. For example, if the parameter of the classifier is specified as `Word w`, then `w.partOfSpeech` may appear as the *`field-access`*. The `cached` keyword instructs the classifier to store the output values it computes in a hash table. As such, the implementations of the `hashCode()` and `equals(Object)` methods in the input type's class play an important role in the behavior of a `cached` classifier. If two input objects return the same value from their `hashCode()` methods and are equivalent according to `equals(Object)`, they will receive the same classification from this classifier.[1]

A cached classifier (using either type of caching) will first check the specified appropriate location to see if a value has already been computed for the given input object. If it has, it is simply returned. Otherwise, the classifier computes the value and stores it in the location before returning it. A discrete classifier will store its value as a `String`. When caching in a field, it will assume that `null` represents the absence of a computed value. A real classifier will store its value as a `double`. When caching in a field, it will assume that `Double.NaN` represents the absence of a computed value. Array returning classifiers will store a value as an array of the appropriate type and assume that `null` represents the absence of a computed value. Generators may not be cached with either keyword. Last but not least, learning classifiers cached with either keyword will not load their (potentially large) internal representations from disk until necessary (i.e., until an object is encountered whose cache location is not filled in). See Section 4.1.2.6 for more information about learning classifiers.

Semantically, every named classifier is a static method. In an LBJ source file, references to classifiers are manipulated and passed to other syntactic constructs, similarly to a functional programming language. The LBJ compiler implements this behavior by storing a classifier's definition in a static method of a Java class of the same name and providing access to that method through objects of that class. As we will see, learning classifiers are capable of modifying their definition, and by the semantics of classifier declarations, these modifications are local to the currently executing process, but not to any particular object. In other words, when the application continues to train a learning classifier on-line, the changes are immediately visible through every object of the classifier's class.

Figure 4.1 gives several examples of classifier declarations. These examples illustrate some key principles LBJ. First, the features produced by a classifier are either discrete or real. If a feature is discrete, the set of allowable values may optionally be specified, contained in curly braces. Any literal values including `int`s, `String`s, and `boolean`s may be used in this set[2].

Next, every classifier takes exactly one object as input and returns one or more features as output. The input object will most commonly be an object from the programmer-designed, object-oriented internal representation of the application domain's data. When the classifier has made its classification(s), it returns one or more features representing those decisions. A complete list of feature return types follows:

[1] Because this type of classifier caching is implemented with a `java.util.WeakHashMap`, it is possible for this statement to be violated if the two objects are not alive in the heap simultaneously. For more information, see the Java API javadoc.

[2] Internally, they'll all be converted to `String`s.

```
discrete{false, true} highRisk(Patient p) <- {
  return p.historyOfCancer() && p.isSmoker();
}

discrete prefix(Word w) cachedin w.prefix <- {
  if (w.spelling > 5) return w.spelling.substring(0, 3);
  return w.spelling;
}

real[] dimensions(Cube c) <- {
  sense c.length;
  sense c.width;
  sense c.height;
}

discrete bigram(Word w) <- spellingTarget && spellingOneAfter
```

Figure 4.1. Classifier declarations declaring hard-coded classifiers.

- `discrete`
- `discrete{` *value-list* `}`
- `real`
- `discrete[]`
- `discrete{` *value-list* `}[]`
- `real[]`
- `discrete%`
- `discrete{` *value-list* `}%`
- `real%`
- `mixed%`

Feature return types ending with square brackets indicate that an array of features is produced by this classifier. The user can expect the feature at a given index of the array to be the same feature with a differing value each time the classifier is called on a different input object. Feature return types ending with a percent sign indicate that this classifier is a *feature generator*. A feature generator may return zero or more features in any order when it is called, and there is no guarantee that the same features will be produced when called on different input objects. Finally, the `mixed%` feature return type indicates that the classifier is a generator of both discrete and real features.

```
discrete learnMe(InputObject o) <-
learn labelingClassifier
  using c1, c2, c3
  from new UserDefinedParser(data)
  with new PreDefinedLearner(parameters)
end
```

Figure 4.2. Learning classifier specification

As illustrated by the fourth classifier in Figure 4.1, a classifier may be composed from other classifiers with classifier operators. The names `spellingTarget` and `spellingOneAfter` here refer to classifiers that were either defined elsewhere in the same source file or that were imported from some other package. In this case, the classifier `bigram` will return a discrete feature whose value is the conjunction of the values of the features produced by `spellingTarget` and `spellingOneAfter`.

All of the classifiers in Figure 4.1 are examples of explicitly coded classifiers. The first three use Java method bodies to compute the values of the returned features. In each of these cases, the names of the returned features are known directly from the classifier declaration's header, so their values are all that is left to compute. In the first two examples, the header indicates that a single feature will be returned. Thus, the familiar `return` statement is used to indicate the feature's value. In the third example, the square brackets in the header indicate that this classifier produces an array of features. Return statements are disallowed in this context since we must return multiple values. Instead, the `sense` statement is used whenever the next feature's value is computed.

For our final example, we will demonstrate the specification of a learning classifier. The `learnMe` learning classifier in Figure 4.2 is supervised by a classifier named `labelingClassifier` whose features will be interpreted as labels of an input training object. Next, after the `using` clause appears a comma separated list of classifier names. These classifiers perform feature extraction. The optional `from` clause designates a parser used to provide training objects to `learnMe` at compile-time. Finally, the optional `with` clause designates a particular learner for `learnMe` to utilize.

4.1.2 Classifier Expressions

As was alluded to above, the right hand side of a classifier declaration is actually a single classifier expression. A classifier expression is one of the following syntactic constructs:

- a classifier name
- a method body (i.e., a list of Java statements in between curly braces)
- a classifier cast expression
- a conjunction of two classifier expressions

- a comma separated list of classifier expressions
- a learning classifier expression
- an inference invocation expression

We have already explored examples of almost all of these. More precise definitions of each follow.

4.1.2.1 Classifier Names

The name of a classifier defined either externally or in the same source file may appear wherever a classifier expression is expected. If the named classifier's declaration is found in the same source file, it may occur anywhere in that source file (in other words, a classifier need not be defined before it is used). If the named classifier has an external declaration it must either be fully qualified (e.g., `myPackage.myClassifier`) or it must be imported by an import declaration at the top of the source file. The class file or Java source file containing the implementation of an imported classifier must exist prior to running the LBJ compiler on the source file that imports it.

4.1.2.2 Method Bodies

A method body is a list of Java statements enclosed in curly braces explicitly implementing a classifier. When the classifier implemented by the method body returns a single feature, the `return` statement is used to provide that feature's value. If the feature return type is `real`, then the `return` statement's expression must evaluate to a `double`. Otherwise, it can evaluate to anything - even an object - and the resulting value will be converted to a `String`. Each method body takes its argument and feature return type from the header of the classifier declaration it is contained in (except when in the presence of a classifier cast expression, discussed in Section 4.1.2.3). For more information on method bodies in LBJ, see Section 4.1.3.

4.1.2.3 Classifier Cast Expressions

When the programmer wishes for a classifier sub-expression on the right hand side of a classifier declaration to be implemented with a feature return type differing from that defined in the header, a classifier cast expression is the solution. For example, the following classifier declaration exhibits a learning classifier (see Section 4.1.2.6) with a real valued feature return type. One of the classifiers it uses as a feature extractor is hard-coded on the fly, but it returns a discrete feature. A classifier cast expression is employed to achieve the desired affect.

```
real dummyClassifier(InputObject o) <-
learn labeler
  using c1, c2, (discrete) { return o.value == 4; }
end
```

Of course, we can see that the hard-coded classifier defined on the fly in this example returns a discrete (`boolean`) value. Without the cast in front of this method body, the LBJ compiler would have assumed it to have a real valued feature return type, and an error would have been

Argument Type	Argument Type	Result Type
discrete	discrete	discrete
discrete	real	real%
discrete	discrete[]	discrete[]
discrete	real[]	real%
discrete	discrete%	discrete%
discrete	real%	real%
discrete	mixed%	mixed%
real	real	real
real	discrete[]	real%
real	real[]	real[]
real	discrete%	real%
real	real%	real%
real	mixed%	mixed%
discrete[]	discrete[]	discrete[]
discrete[]	real[]	real%
discrete[]	discrete%	discrete%
discrete[]	real%	real%
discrete[]	mixed%	mixed%
real[]	real[]	real[]
real[]	discrete%	real%
real[]	real%	real%
real[]	mixed%	mixed%
discrete%	discrete%	discrete%
discrete%	real%	real%
discrete%	mixed%	mixed%
real%	real%	real%
real%	mixed%	mixed%
mixed%	mixed%	mixed%

Table 4.1. Conjunction feature return types given particular argument classifier feature return types.

produced.

When a classifier cast expression is applied to a classifier expression that contains other classifier expressions, the cast propagates down to those classifier expressions recursively as well.

4.1.2.4 Conjunctions

A conjunction is written with the double ampersand operator (`&&`) in between two classifier expressions (see Figure 4.1 for an example). The conjunction of two classifiers results in a new classifier that combines the values of the features returned by its argument classifiers. The nature of the combination depends on the feature return types of the argument classifiers. Table 4.1 enumerates all possibilities and gives the feature return type of the resulting conjunctive classifier.

In general, the following rules apply. Two discrete features are combined simply through concatenation of their values. One discrete and one real feature are combined by creating a new real valued feature whose name is a function of the discrete feature's value and whose value is equal to the real feature's value. When two real features are combined, their values are multiplied.

The conjunction of two classifiers that return a single feature is a classifier returning a single feature. When a classifier returning an array is conjuncted with a either a single feature classifier or a classifier returning an array, the result is an array classifier whose returned array will contain the combinations of every pairing of features from the two argument classifiers. Finally, the conjunction of a feature generator with any other classifier will result in a feature generator producing features representing the combination of every pairing of features from the two argument classifiers.

4.1.2.5 Composite Generators

"Composite generator" is LBJ terminology for a comma separated list of classifier expressions. When classifier expressions are listed separated by commas, the result is a feature generator that simply returns all the features returned by each classifier in the list.

4.1.2.6 Learning Classifier Expressions

Learning classifier expressions have the following syntax:

```
learn [classifier-expression]                  // Labeler
  using classifier-expression                  // Feature extractors
  [from instance-creation-expression [int]]    // Parser
  [with instance-creation-expression]          // Learning algorithm
  [evaluate Java-expression]                   // Alternate eval method
  [cval [int] split-strategy]                  // K-Fold Cross Validation
   [alpha double]                              // Confidence Parameter
   [testingMetric
     instance-creation-expression]]            // Testing Function
  [preExtract boolean]                         // Feature Pre-Extraction
  [progressOutput int]                         // Progress Output Frequency
end
```

The first classifier expression represents a classifier that will provide label features for a supervised learning algorithm. It need not appear when the learning algorithm is unsupervised.[3] The classifier expression in the `using` clause does all the feature extraction on each object, during both training and evaluation. It will often be a composite generator.

The instance creation expression in the `from` clause should create an object of a class that implements the `LBJ2.parser.Parser` interface in the library (see Section 5.4.1). This clause is optional. If it appears, the LBJ compiler will automatically perform training on the learner represented by this learning classifier 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

[3]Keep in mind, however, that LBJ's library currently lacks unsupervised learning algorithm implementations.

`LBJ2.learn.Learner` in the library (see Section 5.2.1).

When the `from` clause appears, the LBJ compiler retrieves objects from the specified parser until it finally returns `null`. One at a time, the feature extraction classifier is applied to each object, and the results are sent to the learning algorithm for processing. However, many learning algorithms perform much better after being given multiple opportunities to learn from each training object. This is the motivation for the integer addendum to this clause. The integer specifies a number of *rounds*, or the number of passes over the training data to be performed by the classifier during training.

The instance creation expression in the `with` clause should create an object of a class derived from the `LBJ2.learn.Learner` class in the library. This clause is also optional. If it appears, the generated Java class implementing this learning classifier will be derived from the class named in the `with` clause. Otherwise, the default learner for the declared return type of this learning classifier will be substituted with default parameter settings.

The `evaluate` clause is used to specify an alternate method for evaluation of the learned classifier. For example, the `SparseNetworkLearner` learner is a multi-class learner that, during evaluation, predicts the label for which it computes the highest score. However, it also provides the `valueOf(Object, java.util.Collection)` method which restricts the prediction to one of the labels in the specified collection. In the application, it's easy enough to call this method in place of `discreteValue(Object)` (discussed in Section 5.1.1), but when this classifier is invoked elsewhere in an LBJ source file, it translates to an invocation of `discreteValue(Object)`. The `evaluate` clause (e.g., `evaluate valueOf(o, MyClass.getCollection())`) changes the behavior of `discreteValue(Object)` (or `realValue(Object)` as appropriate) so that it uses the specified *`Java-expression`* to produce the prediction. Note that *`Java-expression`* will be used only during the evaluation and not the training of the learner specifying the `evaluate` clause.

The `cval` clause enables LBJ's built-in K-fold cross validation system. K-fold cross validation is a statistical technique for assessing the performance of a learned classifier by partitioning the user's set of training data into K subsets such that a single subset is held aside for testing while the others are used for training. LBJ automates this process in order to alleviate the need for the user to perform his own testing methodologies. The optional `split-strategy` argument to the `cval` clause can be used to specify the method with which LBJ will split the data set into subsets (folds). If the `split-strategy` argument is not provided, the default value taken is `sequential`. The user may choose from the following four split strategies:

- `sequential` - The `sequential` split strategy attempts to partition the set of examples into K equally sized subsets based on the order in which they are returned from the user's parser. Given that there are T examples in the data set, the first T/K examples encountered are considered to be the first subset, while the examples between the $(T/K + 1)$'th example and the $(2T/K)$'th example are considered to be the second subset, and so on.
  i.e. [— 1 — | — 2 — | ... | — K —]
- `kth` - The `kth` split strategy also attempts to partition the set of examples in to K equally sized subsets with a round-robin style assignement scheme. The x'th example encountered

is assigned to the $(x\%K)$'th subset.
i.e. [1 2 3 4 ... K 1 2 3 4 ... K ...]

- `random` - The `random` split strategy begins with the assignment given by the `kth` split strategy, and simply mixes the subset assignments. This ensures that the subsets produced are as equally sized as possible.

- `manual` - The user may write their parser so that it returns the unique instance of the `LBJ2.parse.FoldSeparator` class (see the `separator` field) wherever a fold boundary is desired. Each time this object appears, it represents a partition between two folds. Thus, if the k-fold cross validation is desired, it should appear $k-1$ times. The integer provided after the `cval` keyword is ignored and may be omitted in this case.

The `testingMetric` and `alpha` clauses are sub-clauses of `cval`, and, consequently, have no effect when the `cval` clause is not present. The `testingMetric` clause gives the user the opportunity to provide a custom testing methodology. The object provided to the `testingMetric` clause must implement the `LBJ2.learn.TestingMetric` interface. If this clause is not provided, then it will default to the `LBJ2.learn.Accuracy` metric, which simply returns the ratio of correct predictions made by the classifier on the testing fold to the total number of examples contained within said fold.

LBJ's cross validation system provides a confidence interval according to the measurements made by the testing function. With the `alpha` clause, the user may define the width of this confidence interval. The double-precision argument provided to the `alpha` clause causes LBJ to calculate a $(1-a)\%$ confidence interval. For example, "`alpha .07`" causes LBJ to print a 93% confidence interval, according to the testing measurements made. If this clause is not provided, the default value taken is .05, resulting in a 95% confidence interval.

The `preExtract` clause enables or disables the pre-extraction of features from examples. When the argument to this clause is `true`, feature extraction is only performed once, the results of which are recorded in two files. First, an array of all `Feature` objects (see Section 5.1.2) observed during training is serialized and written to a file whose name is the same as the learning classifier's and whose extension is `.lex`. This file is referred to as the *lexicon*. Second, the integer indexes, as they are found in this array, of all features corresponding to each training object are written to a file whose name is the same as the learning classifier's and whose extension is `.ex`. This file is referred to as the *example file*. It is re-read from disk during each training round during both cross validation and final training, saving time when feature extraction is expensive, which is often the case.
If this clause is not provided, the default value taken is `false`.

The `progressOutput` clause defines how often to produce an output message during training. The argument to this clause is an integer which represents the number of examples to process between progress messages. This variable can also be set via a command line parameter using the `-t` option. If a value is provided in both places, the one defined here in the Learning Classifier Expression takes precedence. If no value is provided, then the default value taken is 0, causing progress messages to be given only at the beginning and end of each training pass.

When the LBJ compiler finally processes a learning classifier expression, it generates not only a Java source file implementing the classifier, but also a file containing the results of the computations done during training. This file will have the same name as the classifier but with a `.lc` extension ("lc" stands for "learning classifier"). The directory in which this file and also the lexicon and example files mentioned earlier are written depends on the appearance of certain command line parameters discussed in Section 6.2.

4.1.2.7 Inference Invocations

Inference is the process through which classifiers constrained in terms of each other reconcile their outputs. More information on the specification of constraints and inference procedures can be found in Sections 4.2 and 4.3 respectively. In LBJ, the application of an inference to a learning classifier participating in that inference results in a new classifier whose output respects the inference's constraints. Inferences are applied to learning classifiers via the inference invocation, which looks just like a method invocation with a single argument.

For example, assume that `LocalChunkType` is the name of a discrete learning classifier involved in an inference procedure named `ChunkInference`. Then a new version of `LocalChunkType` that respects the constraints of the inference may be named as follows:

```
discrete ChunkType(Chunk c) <- ChunkInference(LocalChunkType)
```

4.1.3 Method Bodies

Depending on the feature return type, the programmer will have differing needs when designing a method body. If the feature return type is either `discrete` or `real`, then only the value of the single feature is returned through straight forward use of the `return` statement. Otherwise, another mechanism will be required to return multiple feature values in the case of an array return type, or multiple feature names and values in the case of a feature generator. That mechanism is the `sense` statement, described in Section 4.1.3.1.

When a classifier's only purpose is to provide information to a `Learner` (see Section 5.2.1), the `Feature` data type (see Section 5.1.2) is the most appropriate mode of communication. However, in any LBJ source file, the programmer will inevitably design one or more classifiers intended to provide information within the programmer's own code, either in the application or in other classifier method bodies. In these situations, the features' values (and not their names) are the data of interest. Section 4.1.3.2 discusses a special semantics for classifier invocation.

4.1.3.1 The Sense Statement

The `sense` statement is used to indicate that the name and/or value of a feature has been detected when computing an array of features or a feature generator. In these contexts, any number of features may be sensed, and they are returned in the order in which they were sensed.

The syntax of a `sense` statement in an array returning classifier is simply

`sense` *expression* `;`

The expression is interpreted as the value of the next feature sensed. No name need be supplied, as the feature's name is simply the concatenation of the classifier's name with the index this feature will take in the array. This expression must evaluate to a `double` if the method body's feature return type is `real[]`. Otherwise, it can evaluate to anything - even an object - and the resulting value will be converted to a `String`.

The syntax of a `sense` statement in a feature generator is

`sense` *`expression`* `:` *`expression`*`;`

The first expression may evaluate to anything. Its `String` value will be appended to the name of the method body to create the name of the feature. The second expression will be interpreted as that feature's value. It must evaluate to a `double` if the method body's feature return type is `real%`. Otherwise, it can evaluate to anything and the resulting value will be converted to a `String`.

The single expression form of the `sense` statement may also appear in a feature generator method body. In this case, the expression represents the feature's name, and that feature is assumed to be Boolean with a value of `true`.

4.1.3.2 Invoking Classifiers

Under the right circumstances, any classifier may be invoked inside an LBJ method body just as if it were a method. The syntax of a classifier invocation is simply *`name`*`(`*`object`*`)`, where *`object`* is the object to be classified and *`name`* follows the same rules as when a classifier is named in a classifier expression (see Section 4.1.2.1). In general, the semantics of such an invocation are such that the value(s) and not the names of the produced features are returned at the call site.

More specifically:

- A classifier defined to return exactly one feature may be invoked anywhere within a method body. If it has feature return type `discrete`, a `String` will be returned at the call site. Otherwise, a `double` will be returned.
- Classifiers defined to return an array of features may also be invoked anywhere within a method body. Usually, they will return either `String[]` or `double[]` at the call site when the classifier has feature return type `discrete[]` or `real[]` respectively. The only exception to this rule is discussed next.
- When a `sense` statement appears in a method body defined to return an array of features, the lone argument to that `sense` statement may be an invocation of another array returning classifier of the same feature return type. In this case, all of the features returned by the invoked classifier are returned by the invoking classifier, renamed to take the invoking classifier's name and indexes.
- Feature generators may only be invoked when that invocation is the entire expression on the right of the colon in a `sense` statement contained in another feature generator of the same feature return type[4]. In this case, this single `sense` statement will return every

[4] Any feature generator may be invoked in this context in a classifier whose feature return type is `mixed%`.

feature produced by the invoked generator with the following modification. The name of the containing feature generator and the `String` value of the expression on the left of the colon are prepended to every feature's name. Thus, an entire set of features can be translated to describe a different context with a single `sense` statement.

4.1.3.3 Syntax Limitations

When the exact computation is known, LBJ intends to allow the programmer to explicitly define a classifier using arbitrary Java. However, the current version of LBJ suffers from one major limitation. All J2SE 1.4.2 statement and expression syntax is accepted, excluding class and interface definitions. In particular, this means that anonymous classes currently cannot be defined or instantiated inside an LBJ method body.

4.2 Constraints

Many modern applications involve the repeated application of one or more learning classifiers in a coordinated decision making process. Often, the nature of this decision making process restricts the output of each learning classifier on a call by call basis to make all these outputs coherent with respect to each other. For example, a classification task may involve classifying some set of objects, at most one of which is allowed to take a given label. If the learned classifier is left to its own devices, there is no guarantee that this constraint will be respected. Using LBJ's constraint and inference syntax, constraints such as these are resolved automatically in a principled manner.

More specifically, Integer Linear Programming (ILP) is applied to resolve the constraints such that the expected number of correct predictions made by each learning classifier involved is maximized. The details of how ILP works are beyond the scope of this user's manual. See (Punyakanok, Roth, & Yih , 2008) for more details.

This section covers the syntax and semantics of constraint declarations and statements. However, simply declaring an LBJ constraint has no effect on the classifiers involved. Section 4.3 introduces the syntax and semantics of LBJ inference procedures, which can then be invoked (as described in Section 4.1.2.7) to produce new classifiers that respect the constraints.

4.2.1 Constraint Statements

LBJ constraints are written as arbitrary first order Boolean logic expressions in terms of learning classifiers 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 the objects of an internal representation 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).

Each declarative constraint statement contains a single constraint expression and ends in a semicolon. Constraint expressions take one of the following forms:

- An equality predicate *Java-expression* `::` *Java-expression*
- An inequality predicate *Java-expression* `!:` *Java-expression*
- A constraint invocation `@`*name*`(`*Java-expression*`)`
  where the expression must evaluate to an object and *name* follows similar rules as classifier names when they are invoked. In particular, if `MyConstraint` is already declared in `SomeOtherPackage`, it may be invoked with `@SomeOtherPackage.MyConstraint(object)`.
- The negation of an LBJ constraint `!`*constraint*
- The conjunction of two LBJ constraints *constraint* `/\` *constraint*
- The disjunction of two LBJ constraints *constraint* `\/` *constraint*
- An implication *constraint* `=>` *constraint*
- The equivalence of two LBJ constraints *constraint* `<=>` *constraint*
- A universal quantifier `forall (`*type name* `in` *Java-expression*`)` *constraint*
  where the expression must evaluate to a Java `Collection` containing objects of the specified type, and the constraint may be written in terms of *name*.
- An existential quantifier `exists (`*type name* `in` *Java-expression*`)` *constraint*
- An "at least" quantifier
  `atleast` *Java-expression* `of (`*type name* `in` *Java-expression*`)` *constraint*
  where the first expression must evaluate to an an `int`, and the other parameters play similar roles to those in the universal quantifier.
- An "at most" quantifier
  `atmost` *Java-expression* `of (`*type name* `in` *Java-expression*`)` *constraint*

Above, the operators have been listed in decreasing order of precedence. Note that this can require parentheses around quantifiers to achieve the desired effect. For example, the conjunction of two quantifiers can be written like this:

```
(exists (Word w in sentence) someLearner(w) :: "good")
  /\ (exists (Word w in sentence) otherLearner(w) :: "better")
```

The arguments to the equality and inequality predicates are treated specially. If any of these arguments is an invocation of a learning classifier, that classifier and the object it classifies become an inference variable, so that the value produced by the classifier on that object is subject to change by the inference procedure. The values of all other expressions that appear as an argument to either type of predicate are constants in the inference procedure. This includes, in particular, expressions that include a learning classifier invocation as a subexpression. These learning classifier invocations are not treated as inference variables.

4.2.2 Constraint Declarations

An LBJ constraint declaration declares a Java method whose purpose is to locate the objects involved in the inference and generate the constraints. 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:

```
constraint name(type name) method-body
```

where *method-body* may contain arbitrary Java code interspersed with constraint statements all enclosed in curly braces. When invoked with the `@` operator (discussed in Section 4.2.1), the occurrence of a constraint statement in the body of a constraint declaration signifies not that the constraint expression will be evaluated in place, but instead that a first order representation of the constraint expression will be constructed for an inference algorithm to manipulate. The final result produced by the constraint is the conjunction of all constraint statements encountered while executing the constraint.

In addition, constraints declared in this way may also be used as Boolean classifiers as if they had been declared:

```
discrete{"false", "true"} name(type name)
```

Thus, a constraint may be invoked as if it were a Java method (i.e., without the `@` symbol described in Section 4.2.1) anywhere in an LBJ source file, just like a classifier. Such an invocation will evaluate the constraint in place, rather than constructing its first order representation.

4.3 Inference

The syntax of an LBJ inference has the following form:

```
inference name head type name
{
   [type name method-body]+            // "Head-finder" methods
   [[name] normalizedby name ;]*       // How to normalize scores
   subjectto method-body               // Constraints
   with instance-creation-expression   // Names the algorithm
}
```

This structure manages the functions, run-time objects, and constraints involved in an inference. Its header indicates the name of the inference and its *head* parameter. 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. It also need not have the same type as the input parameter of any constraint involved in the inference, although it often will.

After the header, curly braces surround the body of the inference. The body contains the following four elements. First, it contains at least one "head finder" method. Head finder methods are 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 learning classifier

involved in the inference, that learning classifier's input type must have a head finder method in the inference body. Head finder methods are usually very simple. For example:

```
Word w { return w.getSentence(); }
```

might be an appropriate head finder method when the head object has type `Sentence` and one of the classifiers involved in the inference takes `Word`s as input.

Second, the body specifies how the scores produced by each learning classifier should be normalized. The LBJ library contains a set of normalizing functions that may be named here. It is not strictly necessary to use normalization methods, but doing so ensures that the scores computed for each possible prediction may be treated as a probability distribution by the inference algorithm. Thus, we may then reason about the inference procedure as optimizing the expected number of correct predictions.

The syntax of normalizer clauses enables the programmer to specify a different normalization method for each learning classifier involved in the inference. It also allows for the declaration of a default normalizer to be used by learning classifiers which were not given normalizers individually. For example:

```
SomeLearner normalizedby Sigmoid;
normalizedby Softmax;
```

These normalizer clauses written in any order specify that the `SomeLearner` learning classifier should have its scores normalized with the `Sigmoid` normalization method and that all other learning classifiers involved in the inference should be normalized by `Softmax`.

Third, the `subjectto` clause is actually a constraint declaration (see Section 4.2) whose input parameter is the head object. For example, let's say an inference named `MyInference` is declared like this:

```
inference MyInference head Sentence s
```

and suppose also that several other constraints have been declared named (boringly) `Constraint1`, `Constraint2`, and `Constraint3`. Then an appropriate `subjectto` clause for `MyInference` might look like this:

```
subjectto { @Constraint1(s) /\ @Constraint2(s) /\ @Constraint3(s); }
```

The `subjectto` clause may also contain arbitrary Java, just like any other constraint declaration.

Finally, the `with` clause specifies which inference algorithm to use. It functions similarly to the `with` clause of a learning classifier expression (see Section 4.1.2.6).

4.4 "Makefile" Behavior

An LBJ source file also functions as a makefile in the following sense. First, code will only be generated for a classifier definition when it is determined that a change has been made[5] in the

[5]When the file(s) containing the translated code for a given classifier do not exist, this is, of course, also interpreted as a change having been made.

LBJ source for that classifier since the last time the compiler was executed. Second, a learning classifier will only be trained if it is determined that the changes made affect the results of learning. More precisely, any classifier whose definition has changed lexically is deemed "affected". Furthermore, any classifier that makes use of an affected classifier is also affected. This includes method bodies that invoke affected classifiers and conjunctions and learning classifiers involving at least one affected classifier. A learning classifier will be trained if and only if a change has been made to its own source code or it is affected. Thus, when an LBJ source contains many learning classifiers and a change is made, time will not be wasted re-training those that are unaffected.

In addition, the LBJ compiler will automatically compile any Java source files that it depends on, so long as the locations of those source files are indicated with the appropriate command line parameters (see Section 6.2). For example, if the classifiers in an LBJ source file are defined to take classes from the programmer's internal representation as input, the LBJ compiler will automatically compile the Java source files containing those class' implementations if their class files don't already exist or are out of date.

Chapter 5

The LBJ Library

The LBJ programming framework is supported by a library of interfaces, learning algorithms, and implementations of the building blocks described in Chapter 4. This chapter gives a general overview of each of those codes. More detailed usage descriptions can be found in the online Javadoc at `http://flake.cs.uiuc.edu/~rizzolo/LBJ2/library`.

The library is currently organized into five packages. `LBJ2.classify` contains classes related to features and classification. `LBJ2.learn` contains learner implementations and supporting classes. `LBJ2.infer` contains inference algorithm implementations and internal representations for constraints and inference structures. `LBJ2.parse` contains the `Parser` interface and some general purpose internal representation classes. Finally, `LBJ2.nlp` contains some basic natural language processing internal representations and parsing routines. In the future, we plan to expand this library, adding more varieties of learners and domain specific parsers and internal representations.

5.1 `LBJ2.classify`

The most important class in LBJ's library is `LBJ2.classify.Classifier`. This abstract class is the interface through which the application accesses the classifiers defined in the LBJ source file. However, the programmer should, in general, only have need to become familiar with a few of the methods defined there.

One other class that may be of broad interest is the `LBJ2.classify.TestDiscrete` class (discussed in Section 5.1.8), which can automate the performance evaluation of a discrete learning classifier on a labeled test set. The other classes in this package are designed mainly for internal use by LBJ's compiler and can be safely ignored by the casual user. More advanced users who writes their own learners or inference algorithms in the application, for instance, will need to become familiar with them.

5.1.1 `LBJ2.classify.Classifier`

Every classifier declaration in an LBJ source file is translated by the compiler into a Java class that extends this class. When the programmer wants to call a classifier in the application,

he creates an object of his classifier's class using its zero argument constructor and calls an appropriate method on that object. The appropriate method will most likely be one of the following four methods:

`String discreteValue(Object)`:
: This method will only be overridden in the classifier's implementation if its feature return type is `discrete`. Its return value is the value of the single feature this classifier returns.

`double realValue(Object)`:
: This method will only be overridden in the classifier's implementation if its feature return type is `real`. Its return value is the value of the single feature this classifier returns.

`String[] discreteValueArray(Object)`:
: This method will only be overridden in the classifier's implementation if its feature return type is `discrete[]`. Its return value contains the values of all the features this classifier returns.

`double[] realValueArray(Object)`:
: This method will only be overridden in the classifier's implementation if its feature return type is `real[]`. Its return value contains the values of all the features this classifier returns.

There is no method similar to the four above for accessing the values of features produced by a feature generator, since those values are meaningless without their associated names. When the programmer wants access to the actual features produced by any classifier (not just feature generators), the following non-static method is used. Note, however, that the main purpose of this method is for internal use by the compiler.[1]

`FeatureVector classify(Object)`:
: This method is overridden in every classifier implementation generated by the LBJ compiler. It returns a `FeatureVector` which may be iterated through to access individual features (see Section 5.1.3).

Every classifier implementation generated by the compiler overrides the following non-static member methods as well. They provide type information about the implemented classifier.

`String getInputType()`:
: This method returns a `String` containing the fully qualified name of the class this classifier expects as input.

`String getOutputType()`:
: This method returns a `String` containing the feature return type of this classifier. If the classifier is `discrete` and contains a list of allowable values, it will not appear in the output of this method.

[1] One circumstance where the programmer may be interested in this method is to print out the `String` representation of the returned `FeatureVector`.

`String[] allowableValues()`:
: If the classifier is `discrete` and contains a list of allowable values, that list will be returned by this method. Otherwise, an array of length zero is returned. Learners that require a particular number of allowable values may return an array filled with `"*"` whose length indicates that number.

Finally, class `Classifier` provides a simple static method for testing the agreement of two classifiers. It's convenient, for instance, when testing the performance of a learned classifier against an oracle classifier.

`double test(Classifier, Classifier, Object[])`:
: This static method returns the fraction of objects in the third argument that produced the same classifications from the two argument `Classifier`s.

There are several other methods of this class described in the Javadoc documentation. They are omitted here since the programmer is not expected to need them.

5.1.2 `LBJ2.classify.Feature`

This abstract class is part of the representation of the value produced by a classifier. In particular, the name of a feature, but not its value, is stored here. Classes derived from this class (described below) provide storage for the value of the feature. This class exists mainly for internal use by the LBJ compiler, and most programmers will not need to be familiar with it.

`LBJ2.classify.DiscreteFeature`:
: The value of a feature returned by a `discrete` classifier is stored as a `String` in objects of this class.

`LBJ2.classify.DiscreteArrayFeature`:
: The `String` value of a feature returned by a `discrete[]` classifier as well as its integer index into the array are stored in objects of this class.

`LBJ2.classify.RealFeature`:
: The value of a feature returned by a `real` classifier is stored as a `double` in objects of this class.

`LBJ2.classify.RealArrayFeature`:
: The `double` value of a feature returned by a `real[]` classifier as well as its integer index into the array are stored in objects of this class.

5.1.3 `LBJ2.classify.FeatureVector`

`FeatureVector` is a linked-list-style container which stores features that function as labels separately from other features. It contains methods for iterating through the features and labels and adding more of either. Its main function is as the return value of the `Classifier#classify(Object)` method which is used internally by the LBJ compiler (see Section 5.1.1). Most programmers will not need to become intimately familiar with this class.

5.1.4 `LBJ2.classify.Score`

This class represents the `double` score produced by a discrete learning classifier is association with one of its `String` prediction values. Both items are stored in an object of this class. This class is used internally by LBJ's inference infrastructure, which will interpret the score as an indication of how much the learning classifier prefers the associated prediction value, higher scores indicating more preference.

5.1.5 `LBJ2.classify.ScoreSet`

This is another class used internally by LBJ's inference infrastructure. An object of this class is intended to contain one `Score` for each possible prediction value a learning classifier is capable of returning.

5.1.6 `LBJ2.classify.ValueComparer`

This simple class derived from `Classifier` is used to convert a multi-value `discrete` classifier into a Boolean classifier that returns `true` if and only if the multi-valued classifier evaluated to a particular value. `ValueComparer` is used internally by `SparseNetworkLearner` (see Section 5.2.6).

5.1.7 Vector Returners

The classes `LBJ2.classify.FeatureVectorReturner` and `LBJ2.classify.LabelVectorReturner` are used internally by the LBJ compiler to help implement the training procedure when the programmer specifies multiple training rounds (see Section 4.1.2.6). A feature vector returner is substituted as the learning classifier's feature extraction classifier, and a label vector returner is substituted as the learning classifier's labeler (see Section 5.2.1 to see how this substitution is performed). Each of them then expects the object received as input by the learning classifier to be a `FeatureVector`, which is not normally the case. However, as will be described in Section 5.4.4, the programmer may still be interested in these classes if he wishes to continue training a learning classifier for additional rounds on the same data without incurring the costs of performing feature extraction.

5.1.8 `LBJ2.classify.TestDiscrete`

This class can be quite useful to quickly evaluate the performance of a newly learned classifier on labeled testing data. It operates either as a stand-alone program or as a class that may be imported into an application for more tailored use. In either case, it will automatically compute accuracy, precision, recall, and F1 scores for the learning classifier in question.

To use this class inside an application, simply instantiate an object of it using the no-argument constructor. Lets call this object `tester`. Then, each time the learning classifier makes a prediction `p` for an object whose true label is `l`, make the call `tester.reportPrediction(p, l)`. Once all testing objects have been processed, the `printPerformance(java.io.PrintStream)` method may be used print a table of results, or the programmer may make use of the various other methods provided by this class to retrieve the computed statistics. More detailed usage of

all these methods as well as the operation of this class as a stand-alone program is available in the on-line Javadoc.

5.2 LBJ2.learn

The programmer will want to familiarize himself with most of the classes in this package, in particular those that are derived from the abstract class `LBJ2.learn.Learner`. These are the learners that may be selected from within an LBJ source file in association with a learning classifier expression (see Section 4.1.2.6).

5.2.1 LBJ2.learn.Learner

`Learner` is an abstract class extending the abstract class `Classifier` (see Section 5.1.1). It acts as an interface between learning classifiers defined in an LBJ source file and applications that make on-line use of their learning capabilities. The class generated by the LBJ compiler when translating a learning classifier expression will always indirectly extend this class.

In addition to the methods inherited from `Classifier`, this class defines the following non-static, learning related methods. These are not the only methods defined in class `Learner`, and advanced users may be interested in perusing the Javadoc for descriptions of other methods.

`void learn(Object)`:
: The programmer may call this method at any time from within the application to continue the training process given a single example object. The most common use of this method will be in conjunction with a supervised learning algorithm, in which case, of course, the true label of the example object must be accessible by the label classifier specified in the learning classifier expression in the LBJ source file. Note that changes made via this method will not persist beyond the current execution of the application unless the `save()` method (discussed below) is invoked.

`void doneLearning()`:
: Some learning algorithms (usually primarily off-line learning algorithms) save part of their computation until after all training objects have been observed. This method informs the learning algorithm that it is time to perform that part of the computation. When compile-time training is indicated in a learning classifier expression, the LBJ compiler will call this method after training is complete. Similarly, the programmer who performs on-line learning in his application may need to call this method as well, depending on the learning algorithm.

`void forget()`:
: The user may call this method from the application to reinitialize the learning classifier to the state at which it started before any training was performed. Note that changes made via this method will not persist beyond the current execution of the application unless the `save()` method (discussed below) is invoked.

`void save()`:
: As described in Section 4.1.1, the changes made while training a classifier on-line in the

application are immediately visible everywhere in the application. These changes are not written back to disk unless the `save()` method is invoked. Once this method is invoked, changes that have been made from on-line learning will become visible to subsequent executions of applications that invoke this learning classifier.[2]

`LBJ2.classify.ScoreSet scores(Object)`:
This method is used internally by inference algorithms which interpret the scores in the returned `ScoreSet` (see Section 5.1.5) as indications of which predictions the learning classifier prefers and how much they are preferred.

`LBJ2.classify.Classifier getExtractor()`:
This method gives access to the feature extraction classifier used by this learning classifier.

`void setExtractor(LBJ2.classify.Classifier)`:
Use this method to change the feature extraction classifier used by this learning classifier. Note that this change will be remembered during subsequent executions of the application if the `save()` method (described above) is later invoked.

`LBJ2.classify.Classifier getLabeler()`:
This method gives access to the classifier used by this learning classifier to produce labels for supervised learning.

`void setLabeler(LBJ2.classify.Classifier)`:
Use this method to change the labeler used by this learning classifier. Note that this change will be remembered during subsequent executions of the application if the `save()` method (described above) is later invoked.

`void write(java.io.PrintStream)`:
This abstract method must be overridden by each extending learner implementation. A learning classifier derived from such a learner may then invoke this method to produce the learner's internal representation in text form. Invoking this method does *not* make modifications to the learner's internal representation visible to subsequent executions of applications that invoke this learning classifier like the `save()` method does.

In addition, the following *static* flag is declared in every learner output by the LBJ compiler.

`public static boolean isTraining`:
The `isTraining` variable can be used by the programmer to determine if his learning classifier is currently being trained. This ability may be useful if, for instance, a feature extraction classifier for this learning classifier needs to alter its behavior depending on the availability of labeled training data. The LBJ compiler will automatically set this flag `true` during offline training, and it will be initialized `false` in any application using the learning classifier. So, it becomes the programmer's responsibility to make sure it is set appropriately if any additional online training is to be performed in the application.

[2]Please note that the `save()` method currently will not work when the classifier's byte code is packed in a jar file.

5.2.2 `LBJ2.learn.LinearThresholdUnit`

A linear threshold unit is a supervised, mistake driven learner for binary classification. The predictions made by such a learner are produced by computing a score for a given example object and then comparing that score to a predefined threshold. While learning, if the prediction does not match the label, the linear function that produced the score is updated. Linear threshold units form the basis of many other learning techniques.

Class `LinearThresholdUnit` is an abstract class defining a basic API for learners of this type. A non-abstract class extending it need only provide implementations of the following abstract methods.

`void promote(Object)`:
: This method makes an appropriate modification to the linear function when a mistake is made on a positive example (i.e., when the computed score mistakenly fell below the predefined threshold).

`void demote(Object)`:
: This method makes an appropriate modification to the linear function when a mistake is made on a negative example (i.e., when the computed score mistakenly rose above the predefined threshold).

When a learning classifier expression (see Section 4.1.2.6) employs a learner derived from this class, the specified label producing classifier must be defined as `discrete` with a value list containing exactly two values[3]. The learner derived from this class will then learn to produce a higher score when the correct prediction is the second value in the value list.

5.2.3 `LBJ2.learn.SparsePerceptron`

This learner extends class `LinearThresholdUnit` (see Section 5.2.2). It represents its linear function for score computation as a vector of weights corresponding to features. It has an additive update rule, meaning that it promotes and demotes by treating the collection of features associated with a training object as a vector and using vector addition. Finally, parameters such as its learning rate, threshold, the thick separator, and others described in the online Javadoc can be configured by the user.

5.2.4 `LBJ2.learn.SparseAveragedPerceptron`

Extended from `SparsePerceptron` (see Section 5.2.3), this learner computes an approximation of voted Perceptron by averaging the weight vectors obtained after processing each training example. Its configurable parameters are the same as those of `SparsePerceptron`, and, in particular, using this algorithm in conjunction with a positive thickness for the thick separator can be particularly effective.

[3] See Section 4.1.1 for more information on value lists in feature return types.

5.2.5 LBJ2.learn.SparseWinnow

This learner extends class `LinearThresholdUnit` (see Section 5.2.2). It represents its linear function for score computation as a vector of weights corresponding to features. It has a multiplicative update rule, meaning that it promotes and demotes by multiplying an individual weight in the weight vector by a function of the corresponding feature. Finally, parameters such as its learning rates, threshold, and others described in the online Javadoc can be configured by the user.

5.2.6 LBJ2.learn.SparseNetworkLearner

`SparseNetworkLearner` is a multi-class learner, meaning that it can learn to distinguish among two or more discrete label values when classifying an object. It is not necessary to know which label values are possible when employing this learner (i.e., it is not necessary for the label producing classifier specified in a learning classifier expression to be declared with a value list in its feature return type). Values that were never observed during training will never be predicted.

This learner creates a new `LinearThresholdUnit` for each label value it observes and trains each independently to predict `true` when its associated label value is the correct classification. When making a prediction on a new object, it produces the label value corresponding to the `LinearThresholdUnit` producing the highest score. The `LinearThresholdUnit` used may be selected by the programmer, or, if no specific learner is specified, the default is `SparsePerceptron`.

`SparseNetworkLearner` is the default discrete learner; if the programmer does not include a `with` clause in a learning classifier expression (see Section 4.1.2.6) of discrete feature return type, this learner is invoked with default parameters.

5.2.7 LBJ2.learn.NaiveBayes

Naïve Bayes is a multi-class learner that uses prediction value counts and feature counts given a particular prediction value to select the most likely prediction value. It is not mistake driven, as `LinearThresholdUnit`s are. The scores returned by its `scores(Object)` method are directly interpretable as empirical probabilities. It also has a smoothing parameter configurable by the user for dealing with features that were never encountered during training.

5.2.8 LBJ2.learn.StochasticGradientDescent

Gradient descent is a batch learning algorithm for function approximation in which the learner tries to follow the gradient of the error function to the solution of minimal error. This implementation is a stochastic approximation to gradient descent in which the approximated function is assumed to have linear form.

`StochasticGradientDescent` is the default real learner; if the programmer does not include a `with` clause in a learning classifier expression (see Section 4.1.2.6) of real feature return type, this learner is invoked with default parameters.

5.2.9 LBJ2.learn.Normalizer

A normalizer is a method that takes a set of scores as input and modifies those scores so that they obey particular constraints. Class `Normalizer` is an abstract class with a single abstract method `normalize(LBJ2.classify.ScoreSet)` (see Section 5.1.5) which is implemented by extending classes to define this "normalization." For example:

`LBJ2.learn.Sigmoid`:
This `Normalizer` simply replaces each score s_i in the given `ScoreSet` with $\frac{1}{1+e^{s_i}}$. After normalization, each score will be greater than 0 and less than 1.

`LBJ2.learn.Softmax`:
This `Normalizer` replaces each score with the fraction of its exponential out of the sum of all scores' exponentials. More precisely, each score s_i is replaced by $\frac{e^{s_i}}{\sum_j e^{s_j}}$. After normalization, each score will be positive and they will sum to 1.

`LBJ2.learn.IdentityNormalizer`:
This `Normalizer` simply returns the same scores it was passed as input.

5.2.10 LBJ2.learn.WekaWrapper

The `WekaWrapper` class is meant to wrap instances of learners from the WEKA library of learning algorithms[4]. The `LBJ2.learn.WekaWrapper` class converts between the internal representations of LBJ and WEKA on the fly, so that the more extensive set of algorithms contained within WEKA can be applied to projects written in LBJ.

The `WekaWrapper` class extends `LBJ2.learn.Learner`, and carries all of the functionality that can be expected from a learner. A standard invocation of `WekaWrapper` could look something like this:

```
new WekaWrapper(new weka.classifiers.bayes.NaiveBayes())
```

Restrictions

- It is crucial to note that WEKA learning algorithms do not learn online. Therefore, whenever the `learn` method of the `WekaWrapper` is called, no learning actually takes place. Rather, the input object is added to a collection of examples for the algorithm to learn once the `doneLearning()` method is called.
- The `WekaWrapper` only supports features which are either discrete without a value list, discrete with a value list, or real. In WEKA, these correspond to `weka.core.Attribute` objects of type `String`, `Nominal`, and `Numerical`. In particular, array producing classifiers and feature generators may not be used as features for a learning classifier learned with this class. See section 4.1.1 for further discussion Classifier Declarations.
- When designing a learning classifier which will use a learning algorithm from WEKA, it is important to note that very very few algorithms in the WEKA library support `String`

[4] http://www.cs.waikato.ac.nz/ml/weka/

attributes. In LBJ, this means that it will be very hard to find a learning algorithm which will learn using a `discrete` feature extractor which does not have a value list. I.e. value lists should be provided for discrete feature extracting classifiers whenever possible.

- Feature pre-extraction must be enabled in order to use the `WekaWrapper` class. Feature pre-extraction is enabled by using the `preExtract` clause in the `LearningClassifierExpression` (discussed in 4.1.2.6).

5.3 LBJ2.infer

The `LBJ2.infer` package contains many classes. The great majority of these classes form the internal representation of both propositional and first order constraint expressions and are used internally by LBJ's inference infrastructure. Only the programmer who designs his own inference algorithm in terms of constraints needs to familiarize himself with these classes. Detailed descriptions of them are provided in the Javadoc.

There are a few classes, however, that are of broader interest. First, the `Inference` class is an abstract class from which all inference algorithms implemented for LBJ are derived. It is described below along with the particular algorithms that have already been implemented. Finally, the `InferenceManager` class is used internally by the LBJ library when applications using inference are running.

5.3.1 LBJ2.infer.Inference

`Inference` is an abstract class from which all inference algorithms are derived. Executing an inference generally evaluates all the learning classifiers involved on the objects they have been applied to in the constraints, as well as picking new values for their predictions so that the constraints are satisfied. An object of this class keeps track of all the information necessary to perform inference in addition to the information produced by it. Once that inference has been performed, constrained classifiers access the results through this class's interface to determine what their constrained predictions are. This is done through the `valueOf(LBJ2.learn.Learner, Object)` method described below.

`String valueOf(LBJ2.learn.Learner, Object)`:
: The arguments to this method are objects representing a learning classifier and an object involved in the inference. Calling this method causes the inference algorithm to run, if it has not been run before. This method then returns the new prediction corresponding to the given learner and object after constraints have been resolved.

5.3.2 LBJ2.infer.GLPK

This inference algorithm, which may be named in the `with` clause of the LBJ `inference` syntax, uses Integer Linear Programming (ILP) to maximize the expected number of correct predictions while respecting the constraints. Upon receiving the constraints represented as First Order Logic (FOL) formulas, this implementation first translates those formulas to a propositional representation. The resulting propositional expression is then translated to a set of linear inequalities by recursively translating subexpressions into sets of linear inequalities that bound newly created

variables to take their place.

The number of linear inequalities and extra variables generated is linear in the depth of the tree formed by the propositional representation of the constraints. This tree is not binary; instead, nodes representing operators that are associative and commutative such as conjunction and disjunction have multiple children and are not allowed to have children representing the same operator (i.e., when they do, they are collapsed into the parent node). So both the number of linear inequalities and the number of extra variables created will be relatively low. However, the performance of any ILP algorithm is very sensitive to both these numbers, since ILP is NP-hard. On a 3 Ghz machine, the programmer will still do well to keep both these numbers under 20,000 for any given instance of the inference problem.

The resulting ILP problem is then solved by the GNU Linear Programming Kit (GLPK), a linear programming library written in C.[5] This software must be downloaded and installed separately before installing LBJ, or the `GLPK` inference algorithm will be disabled. If LBJ has already been installed, it must be reconfigured and reinstalled (see Chapter 6.1) after installing GLPK.

5.4 LBJ2.parse

This package contains the very simple `Parser` interface, implementers of which are used in conjunction with learning classifier expressions in an LBJ source file when off-line training is desired (see Section 4.1.2.6). It also contains some general purpose internal representations which may be of interest to a programmer who has not yet written the internal representations or parsers for the application.

5.4.1 LBJ2.parse.Parser

The LBJ compiler is capable of automatically training a learning classifier given training data, so long as that training data comes in the form of objects ready to be passed to the learner's `learn(Object)` method. Any class that implements the `Parser` interface can be utilized by the compiler to provide those training objects. This interface simply consists of a single method for returning another object:

`Object next()`:
: This is the only method that an implementing class needs to define. It returns the next training `Object` until no more are available, at which point it returns `null`.

5.4.2 LBJ2.parse.LineByLine

This abstract class extends `Parser` but does not implement the `next()` method. It does, however, define a constructor that opens the file with the specified name and a readLine() method that fetches the next line of text from that file. Exceptions (as may result from not being able to open or read from the file) are automatically handled by printing an error message and exiting the application.

[5] http://www.gnu.org/software/glpk/

5.4.3 LBJ2.parse.ChildrenFromVectors

This parser calls a user specified, `LinkedVector` (see Section 5.4.6) returning `Parser` internally and returns the `LinkedChild`ren (see Section 5.4.5) of that vector one at a time through its `next()` method. One notable `LinkedVector` returning `Parser` is `LBJ2.nlp.WordSplitter` discussed in Section 5.5.2.

5.4.4 LBJ2.parse.FeatureVectorParser

This parser is used internally by the LBJ compiler (and may be used by the programmer as well) to continue training the learning classifier after the first round of training without incurring the cost of feature extraction. See Section 4.1.2.6 for more information on LBJ's behavior when the programmer specifies multiple training rounds. That section describes how lexicon and example files are produced, and these files become the input to `FeatureVectorParser`.

The objects produced by `FeatureVectorParser` will be `FeatureVector`s, which are not normally the input to any classifier, including the learning classifier we'd like to continue training. So, the programmer must first replace the learning classifier's feature extractor with a `FeatureVectorReturner` and its labeler with a `LabelVectorReturner` (see Section 5.1.7) before calling `learn(Object)`. After the new training objects have been exhausted, the original feature extractor and labeler must be restored before finally calling `save()`.

For example, if a learning classifier named `MyTagger` has been trained for multiple rounds by the LBJ compiler, the lexicon and example file will be created with the names `MyTagger.lex` and `MyTagger.ex` respectively. Then the following code in an application will continue training the classifier for an additional round:

```
MyTagger tagger = new MyTagger();
Classifier extractor = tagger.getExtractor();
tagger.setExtractor(new FeatureVectorReturner());
Classifier labeler = tagger.getLabeler();
tagger.setLabeler(new LabelVectorReturner());
FeatureVectorParser parser =
   new FeatureVectorParser("MyTagger.ex", "MyTagger.lex");
for (Object vector = parser.next(); vector != null; vector = parser.next())
   tagger.learn(vector);
tagger.setExtractor(extractor);
tagger.setLabeler(labeler);
tagger.save();
```

5.4.5 LBJ2.parse.LinkedChild

Together with `LinkedVector` discussed next, these two classes form the basis for a simple, general purpose internal representation for raw data. `LinkedChild` is an abstract class containing pointers to two other `LinkedChild`ren, the "previous" one and the "next" one. It may also store a pointer to its parent, which is a `LinkedVector`. Constructors that set up all these links are also provided, simplifying the implementation of the parser.

5.4.6 LBJ2.parse.LinkedVector

A LinkedVector contains any number of LinkedChildren and provides random access to them in addition to the serial access provided by their links. It also provides methods for insertion and removal of new children. A LinkedVector is itself also a LinkedChild, so that hierarchies are easy to construct when sub-classing these two classes.

5.5 LBJ2.nlp

The programmer of Natural Language Processing (NLP) applications may find the internal representations and parsing algorithms implemented in this package useful. There are representations of words, sentences, and documents, as well as parsers of some common file formats and algorithms for word and sentence segmentation.

5.5.1 Internal Representations

These classes may be used to represent the elements of a natural language document.

LBJ2.nlp.Word:
: This simple representation of a word extends the LinkedChild class (see Section 5.4.5) and has space for its spelling and part of speech tag.

LBJ2.nlp.Sentence:
: Objects of the Sentence class store only the full text of the sentence in a single String. However, a method is provided to heuristically split that text into Word objects contained in a LinkedVector.

LBJ2.nlp.NLDocument:
: Extended from LinkedVector, this class has a constructor that takes the full text of a document as input. Using the methods in Sentence and SentenceSplitter, it creates a hierarchical representation of a natural language document in which Words are contained in LinkedVectors representing sentences which are contained in this LinkedVector.

LBJ2.nlp.POS:
: This class may be used to represent a part of speech, but it used more frequently to simply retrieve information about the various parts of speech made standard by the Penn Treebank project (Marcus, Santorini, & Marcinkiewicz, 1994).

5.5.2 Parsers

The classes listed in this section are all derived from class LineByLine (see Section 5.4.2). They all contain (at least) a constructor that takes a single String representing the name of a file as input. The objects they return are retrieved through the overridden next() method.

LBJ2.nlp.SentenceSplitter:
: Use this Parser to separate sentences out from plain text. The class provides two constructors, one for splitting sentences out of a plain text file, and the other for splitting sentences out of plain text already stored in memory in a String[]. The user can then retrieve

`Sentence`s one at a time with the `next()` method, or all at once with the `splitAll()` method. The returned `Sentence`s' start and end fields represent offsets into the text they were extracted from. Every character in between those two offsets inclusive, including extra spaces, newlines, etc., is included in the `Sentence` as it appeared in the paragraph.[6]

`LBJ2.nlp.WordSplitter`:
This parser takes the plain, unannotated `Sentence`s (see Section 5.5.1) returned by another parser (e.g., `SentenceSplitter`) and splits them into Word objects. Entire sentences now represented as `LinkedVector`s (see Section 5.4.6) are then returned one at a time by calls to the `next()` method.

`LBJ2.nlp.ColumnFormat`:
This parser returns a `String[]` representing the rows of a file in column format. The input file is assumed to contain fields of non-whitespace characters separated by any amount of whitespace, one line of which is commonly used to represent a word in a corpus. This parser breaks a given line into one `String` per field, omitting all of the whitespace. A common usage of this class will be in extending it to create a new `Parser` that calls `super.next()` and creates a more interesting internal representation with the results.

`LBJ2.nlp.POSBracketToVector`:
Use this parser to return `LinkedVector` objects representing sentences given file names of POS bracket form files to parse. These files are expected to have one sentence per line, and the format of each line is as follows:

$$\texttt{(}\texttt{pos}_1\ \texttt{spelling}_1\texttt{)}\ \texttt{(}\texttt{pos}_2\ \texttt{spelling}_2\texttt{)}\ \ldots\ \ \texttt{(}\texttt{pos}_n\ \texttt{spelling}_n\texttt{)}$$

It is also expected that there will be exactly one space between a part of speech and the corresponding spelling and between a closing parenthesis and an opening parenthesis.

[6] If the constructor taking a `String[]` as an argument is used, newline characters are inserted into the returned sentences to indicate transitions from one element of the array to the next.

Chapter 6

Installation and Command Line Usage

6.1 Installation

LBJ is written entirely in Java - almost. The Java Native Interface (JNI) is utilized to interface with the GNU Linear Programming Kit (GLPK) which is used to perform inference (see Section 5.3.2), requiring a small amount of C to complete the connection. This C code must be compiled as a library so that it can be dynamically linked to the JVM at run-time in any application that uses inference. Thus, the GNU Autotools became a natural choice for LBJ's build system. More information on building and installing LBJ from its source code is presented below.

On the other hand, some users' applications may not require LBJ's automated inference capabilities. In this case, installation is as easy as downloading two jar files from the Cognitive Computation Group's website[1] and adding them to your `CLASSPATH` environment variable. `LBJ2.jar` contains the classes implementing the LBJ compiler. `LBJ2Library.jar` contains the library classes. If this is your chosen method of installation, you may safely skip to the section on command line usage below.

Alternatively, the source code for both the compiler and the library can be downloaded from the same web site. Download the file `lbj-2.x.x.tar.gz` and unpack it with the following command:

```
tar zxf lbj-2.x.x.tar.gz
```

The `lbj-2.x.x` directory is created, and all files in the package are placed in that directory. Of particular interest is the file `configure`. This is a shell script designed to automatically detect pertinent parameters of your system and to create a set of makefiles that builds LBJ with respect to those parameters. In particular, this script will detect whether or not you have GLPK installed. If you do, LBJ will be compiled with inference enabled.[2] The `configure` script itself was built automatically by the GNU Autotools, but you will *not* need them installed on your system to make use of it.

[1] `http://l2r.cs.uiuc.edu/~cogcomp`

[2] GLPK is a separate software package that must be downloaded, compiled, and installed before LBJ is configured in order for LBJ to make use of it. Download it from `http://www.gnu.org/software/glpk/`

By default, the `configure` script will create makefiles that intend to install LBJ's JNI libraries and headers in system directories such as `/usr/local/lib` and `/usr/local/include`. If you have root privileges on your system, this will work just fine. Otherwise, it will be necessary to use `configure`'s `--prefix` command line option. For example, running `configure` with `--prefix=$HOME` will create makefiles that install LBJ's libraries and headers in similarly named subdirectories of your user account's root directory, such as `~/lib` and `~/include`. The `configure` script has many other options as well. Use `--help` on the command line for more information.

If you choose to use the `--prefix` command line option, then it is a reasonable assumption that you also used it when building and installing GLPK. In that case, the following environment variables must be set *before* running LBJ's `configure` script. `CPPFLAGS` is used to supply command line parameters to the C preprocessor. We will use it to add the directory where the GLPK headers were installed to the include path. `LDFLAGS` is used to supply command line parameters to the linker. We will use it to add the directory where the GLPK library was installed to the list of paths that the linker will search in. For example, in the `bash` shell:

```
export CPPFLAGS=-I$HOME/include
export LDFLAGS=-L$HOME/lib
```

or in `csh`:

```
setenv CPPFLAGS -I${HOME}/include
setenv LDFLAGS -L${HOME}/lib
```

The last step in making sure that inference will be enabled is to ensure that the file `jni.h` is on the include path for the C preprocessor. This file comes with your JVM distribution. It is often installed in a standard location already, but if it isn't, we must set `CPPFLAGS` in such a way that it adds all the paths we are interested in to the include path. For example, in the `bash` shell:

```
export JVMHOME=/usr/lib/jvm/java-6-sun
export CPPFLAGS="$CPPFLAGS -I$JVMHOME/include"
export CPPFLAGS="$CPPFLAGS -I$JVMHOME/include/linux"
```

or in `csh`:

```
setenv JVMHOME /usr/lib/jvm/java-6-sun
setenv CPPFLAGS "${CPPFLAGS} -I${JVMHOME}/include"
setenv CPPFLAGS "${CPPFLAGS} -I${JVMHOME}/include/linux"
```

At long last, we are ready to build and install LBJ with the following command:

```
./configure --prefix=$HOME && make && make install
```

If all goes well, you will see a message informing you that a library has been installed and that certain extra steps may be necessary to ensure that this library can be used by other programs. Follows these instructions. Also, remember to add the `lbj-2.x.x` directory to your `CLASSPATH` environment variable.

LBJ's makefile also contains rules for creating the jars that are separately downloadable from the website and for creating the Javadoc documentation for both compiler and library. To create

the jars, simply type `make jars`. To create the Javadoc documentation, you must first set the environment variable `LBJ2_DOC` equal to the directory in which you would like the documentation created. Then type `make doc`.

Finally, users of the VIM editor may be interested in `lbj.vim`, the LBJ syntax highlighting file provided in the tar ball. If you have not done so already, create a directory named `.vim` in your home directory. In that directory, create a file named `filetype.vim` containing the following text:

```
if exists("did_load_filetypes")
  finish
endif
augroup filetypedetect
  au! BufRead,BufNewFile *.lbj         setf lbj
augroup END
```

Then create the subdirectory `.vim/syntax` and place the provided `lbj.vim` file in that subdirectory. Now, whenever VIM edits a file whose extension is `.lbj`, LBJ syntax highlighting will be enabled.

6.2 Command Line Usage

The LBJ compiler is itself written in Java. It calls `javac` both to compile classes that its source file depends on and to compile the code it generates. Its command line usage is as follows:

```
java LBJ2.Main [options] <source file>
```

where `[options]` is zero or more of the following:

`-c`	Compile only: This option tells LBJ2 to translate the given source to Java, but not to compile the generated Java sources or do any training.
`-d <dir>`	Any class files generated during compilation will be written in the specified directory, just like javac's -d command line parameter.
`-j <a>`	Sends the contents of `<a>` to `javac` as command line arguments while compiling. Don't forget to put quotes around `<a>` if there is more than one such argument or if the argument has a parameter.
`-t <n>`	Enables progress output during training of learning classifiers. A message containing the date and time will be printed to `STDOUT` after every `<n>` training objects have been processed.
`-v`	Prints the version number and exits.
`-w`	Disables the output of warning messages.
`-x`	Clean: This option deletes all files that would have been generated otherwise. No new code is generated, and no training takes place.
`-gsp <dir>`	LBJ will potentially generate many Java source files. Use this option to have LBJ write them to the specified directory instead of the current directory. `<dir>` must already exist. Note that LBJ will also compile these files which can result in even more class files than there were sources. Those class files will also be written in `<dir>` unless the `-d` command line parameter is utilized as well.
`-sourcepath <dir>`	If the LBJ source depends on classes whose source files cannot be found on the user's classpath, specify the directories where they can be found using this parameter. It works just like javac's `-sourcepath` command line parameter.
`--parserDebug`	Debug: This option enables debugging output during parsing.
`--lexerOutput`	Lexer output: With this option enabled, the lexical token stream will be printed, after which the compiler will terminate.
`--parserOutput`	Parser output: With this option enabled, the parsed abstract syntax tree will be printed, after which the compiler will quit.
`--semanticOutput`	Semantic analysis output: With this option enabled, some information computed by semantic analysis will be printed, after which the compiler will quit.

By default, all files generated by LBJ will be created in the same directory in which the LBJ source file is found. To place generated Java sources in a different directory, use the `-gsp` (or `-generatedsourcepath`) command line option. The lexicon and example files described in Section 4.1.2.6 are also placed in the directory specified by this option. In addition, the generated sources' class files will be created in that directory unless the `-d` command line option is also specified. This option places all generated class files in the specified directory, just like `javac`'s `-d` option. The "learning classifier" file with extension `.lc` (also discussed in Section 4.1.2.6) will also be placed in the directory specified by the `-d` option. Another option similar to `javac` is the `-sourcepath` option for specifying extra directories in which Java source files are found. Both the `-d` and `-sourcepath` options should be given directly to LBJ if they are given at all. Do not specify them inside LBJ's `-j` option. Finally, LBJ does not offer a `-classpath` option. Simply give this parameter to the JVM instead.

For example, say an employee of the XYZ company is building a new software package called ABC with the help of LBJ. This is a large project, and compiling the LBJ source file will generate many new Java sources. She places her LBJ source file in a new working directory along side three new subdirectories: `src`, `class`, and `lbj`.

```
$ ls
abc.lbj  src/    class/  lbj/
```

Next, since all the source files in the ABC application will be part of the `com.xyz.abc` package, she creates the directory structure `com/xyz/abc` as a subdirectory of the `src` directory. Application source files are then placed in the `src/com/xyz/abc` directory. Next, at the top of her LBJ source file she writes the line `package com.xyz.abc;`. Now she is ready to run the following commands:

```
$ java -cp $CLASSPATH:class LBJ2.Main -sourcepath src -gsp lbj -d class abc.lbj
. . .
$ javac -classpath $CLASSPATH:class -sourcepath lbj:src -d class \
        src/com/xyz/abc/*.java
$ jar cvf abc.jar -C class com
```

The first command creates the `com/xyz/abc` directory structure in both of the `lbj` and `class` directories. LBJ then generates new Java sources in the `lbj/com/xyz/abc` directory and class files in the `class/com/xyz/abc` directory. Now that the necessary classifiers' implementations exist, the second command compiles the rest of the application. Finally, the last command prepares a jar file containing the entire ABC application. Users of ABC need only add `abc.jar` to their `CLASSPATH`.

There are two other JVM command line parameters that will be of particular interest to programmers working with large datasets. Both increase the amount of memory that Java is willing to utilize while running. The first is `-Xmx<size>` which sets the maximum Java heap size. It should be set as high as possible, but not so high that it causes page-faults for the JVM or for some other application on the same computer. This value must be a multiple of 1024 greater than 2MB and can be specified in kilobytes (`K`, `k`), megabytes (`M`, `m`), or gigabytes (`G`, `g`).

The second is `-XX:MaxPermSize=<size>` which sets the maximum size of the *permanent gen-*

eration. This is a special area of the heap which stores, among other things, canonical representations for the `String`s in a Java application. Since a learned classifier can contain many `String`s, it may be necessary to set it higher than the default of 64 MB. For more information about the heap and garbage collection, see `http://java.sun.com/docs/hotspot/gc5.0/gc_tuning_5.html`.

With these two command line parameters, a typical LBJ compiler command line might look like:

```
java -Xmx512m -XX:MaxPermSize=512m LBJ2.Main Test.lbj
```

When it is necessary to run the compiler with these JVM settings, it will also be necessary to run the application that uses the generated classifiers with the same or larger settings.

Chapter 7

Licenses and Copyrights

The LBJ compiler and library are covered by the University of Illinois Open Source License. The LBJ compiler contains code generated by the JLex automatic scanner generator and the Java CUP automatic parser generator, and it is packaged with run-time classes from the CUP distribution.

7.1 LBJ's License

Developed by:
The Cognitive Computations Group
Department of Computer Science
University of Illinois at Urbana-Champaign
http://l2r.cs.uiuc.edu/~cogcomp

- Neither the names of the Cognitive Computations Group, nor the University of Illinois at Urbana-Champaign, nor the names of its contributors may be used to endorse or promote products derived from this Software without specific prior written permission.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE CONTRIBUTORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS WITH THE SOFTWARE.

7.2 JLex's License

Copyright 1996-2003 by Elliot Joel Berk and C. Scott Ananian

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both the copyright notice and this permission notice and warranty disclaimer appear in supporting documentation, and that the name of the authors or their employers not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

The authors and their employers disclaim all warranties with regard to this software, including all implied warranties of merchantability and fitness. In no event shall the authors or their employers be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of this software.

This is an open source license. It is also GPL-Compatible (see entry for "Standard ML of New Jersey"). The portions of JLex output which are hard-coded into the JLex source code are (naturally) covered by this same license.

Java is a trademark of Sun Microsystems, Inc. References to the Java programming language in relation to JLex are not meant to imply that Sun endorses this product.

7.3 CUP's License

CUP Parser Generator Copyright Notice, License, and Disclaimer
Copyright 1996-1999 by Scott Hudson, Frank Flannery, C. Scott Ananian

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both the copyright notice and this permission notice and warranty disclaimer appear in supporting documentation, and that the names of the authors or their employers not

be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

The authors and their employers disclaim all warranties with regard to this software, including all implied warranties of merchantability and fitness. In no event shall the authors or their employers be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of this software.

This is an open source license. It is also GPL-Compatible (see entry for "Standard ML of New Jersey"). The portions of CUP output which are hard-coded into the CUP source code are (naturally) covered by this same license, as is the CUP runtime code linked with the generated parser.

Java is a trademark of Sun Microsystems, Inc. References to the Java programming language in relation to CUP are not meant to imply that Sun endorses this product.

References

[Marcus, Santorini, & Marcinkiewicz , 1994] Marcus, M. P.; Santorini, B.; and Marcinkiewicz, M. A. 1994. Building a large annotated corpus of english: The penn treebank. *Computational Linguistics* 19(2):313–330.

[Punyakanok, Roth, & Yih , 2008] Punyakanok, V.; Roth, D.; and Yih, W. 2008. The Importance of Syntactic Parsing and Inference in Semantic Role Labeling. *Computational Linguistics* 34(2):257–287.