CS 288: Statistical NLP
Assignment 4: Parsing and Structured Prediction

Due 5/09/11

In this assignment, you will build an English treebank parser. You will consider the problem of learning a grammar from a treebank (both generatively and discriminatively) and the problem of parsing with that grammar.

Setup: The starting class for this assignment is

```
edu.berkeley.nlp.assignments.PCFGParserTester
```

Make sure you can access the source and data files.

Description: In this project, you will build a broad-coverage parser. You may either build an agenda-driven PCFG parser, or an array-based CKY parser. We will first go over the data flow, then describe the support classes that are provided.

Currently, files 200 to 2199 of the Treebank are read in as training data, as is standard for this data set. Depending on whether you run with `-validate` or `-test`, either files 2200 to 2299 are read in (validation), or 2300 to 2399 are read in (test). You can look in the data directory if you're curious about the native format of these files, but all I/O is taken care of by the provided code. You can always run on fewer training or test files to speed up your preliminary experiments, especially while debugging (where you might first want to train and test on the same, single file, or even just a single tree).

Once the trees are read, the code constructs a `BaselineParser`, which implements the `Parser` interface (with only one method: `getBestParse()`). The parser is then used to predict trees for the sentences in the test set. You can control the maximum length of training and testing sentences with the parameters `-maxTrainLength` and `-maxTestLength`. The standard setup is to train on sentences of all lengths and test on sentences of length less than or equal to 40 words. Your final parser should work on sentences of at least length 20 in a reasonable time (5 seconds per 20-word sentence should be achievable without too much optimization; a good research parser will parse a 20 word sentence in more like 0.1 seconds).

Sentences are evaluated using F1 over labeled spans (see `EnglishPennTreebankParseEvaluator`). Note that tags are *not* included in this evaluation.

This baseline parser is quite terrible - it takes a sentence, tags each word with its most likely tag (i.e. runs a unigram tagger), then looks for that exact tag sequence in the training set. If it finds an exact match, it answers with a known parse for that tag sequence. If no match is found, it constructs a right-branching tree, with nodes' labels chosen independently, conditioned only on the length of the span of a node. This baseline is just a crazy placeholder — you're going to provide a better solution.

You will be implementing three different parsing models. You will provide implementations of the `ParserFactory` interface in `GenerativeParserFactory`, `DiscriminativeParserFactory`, and `AwesomeParserFactory`. These factories construct (and train) a parser given: a list of training trees; an `Indexer` which assigns indices to features; and an instance of `Weights` which manages weights for the grammar and lexicon. Note that the weights are *not* set to anything useful by default, and part of the work of your parser will be in setting these weights.

You should familiarize yourself with these basic classes:

`Tree`	CFG tree structures, (pretty-print with `Trees.PennTreeRenderer`)
`UnaryRule/BinaryRule/Grammar/Lexicon`	CFG rules and accessors
`Weights`	Manages weights (scores) of grammar rules

You will be using these classes no matter what kind of parser you build. If you choose to build an agenda-based parser, you will find `util.GeneralPriorityQueue` useful. If you choose to build an array-based CKY parser, you will instead find the `UnaryClosure` class useful. It computes the reflexive, transitive closure of the unary subset of a grammar, and also maps closure rules to their best backing paths. Note that very little of the code we provide you is required by the harness, so you can implement substitutes for `Grammar`, `Lexicon`, etc.

The `Grammar` implements a basic PCFG grammar with some self-explanatory accessors. An instance of `Grammar` can be constructed by calling the static factory method `generativeGrammarFromTrees`. This method takes a (binarized) treebank as input, and sets the weights of all unary and binary rules in the treebank to their log relative frequency count.

The `Lexicon` class provides basic functionality for scoring POS tags for words. The `Lexicon` class takes a treebank and collects some simple counts. It also takes an implementation of `LexiconFeaturizer`, which is responsible for extracting features on (word, tag) pairs. We have provide a basic implementation in `BasicLexiconFeaturizer`. This lexicon is minimal, but handles rare and unknown words adequately for the present purposes. Note that where the `Grammar` extracts indicator features on all context-free productions, this lexicon extracts a *single*, real-valued feature representing an approximate tagging probability. Don't forget to set the weight of this feature to something reasonable (probably 1.0 for a generative parser).

The `Weights` class manages weights (scores) of features for both the lexicon and grammar. The weights are represented internally as a `double[]` array, with indices in the array referring to indices in an instance of `Indexer<FeatureIndexable>`. `FeatureIndexable` is just a marker interface which marks objects that can be indexed by the feature indexer. Any object which implements `FeatureIndexable` should implement `equals` and `hashCode` so that occurences of the same feature get mapped to the same index. In general, you should subclass `FeatureIndexable` for each

feature "template" you use. See `ExampleLexiconFeaturizer.FirstLetterFeatureTemplate` for an example of a simple feature template.

At this point, you should manually scan through a few of the training trees to get a sense of the format and range of inputs. Something you'll notice is that the grammar has relatively few non-terminal symbols (27 plus part-of-speech tags) but thousands of rules, many trinary-branching or longer. As we discussed in class, most parsers (including yours) require grammars in which the rules are at most binary branching. You can binarize and unbinarize trees using the `TreeAnnotations` class. The default implementation binarizes the trees in a way that doesn't generalize the n-ary grammar at all (convince yourself of this). You should run some trees through the binarization process and look at the results to get an idea of what's going on. If you annotate/binarize the training trees, you should be able to construct a PCFG (represented by a `Grammar` and `Lexicon`) out of them. Your goal is to build a parser which parses novel test sentences using that PCFG.

Note that if you fail to parse a test sentence (because your grammar does not permit any valid trees), then the test harness will still expect you to return a non-null that has a `ROOT` tag and at least one child. A simple solution is to return:

```
new Tree<String>("ROOT", Collections.singletonList(new Tree<String>("JUNK")))
```

Once you've got a parser that, given a test sentence, returns a parse of that sentence using the training grammar and a reasonable binarization scheme, you have all you need to get a decent generative parser. With a 2nd-order / 2nd-order grammar, meaning using parent annotation (symbols like `NP^S` instead of `NP`) and forgetful binarization (symbols like `@VP->...NP_PP` which abstract the horizontal history, instead of `@VP->_VBD_RB_NP_PP` which record the entire history), you should be able to get around 84.0 F1 with `-maxTrainLength` and `-maxTestLength` set to 15.

The next step will be implementing some kind of discriminative training (in `DiscriminativeParserFactory`). You should either train a CRF using gradient descent or LBFGS (see `LBFGSMinimizer`), or you can train a Max-Margin Markov Network using either the algorithm in Ben Taskar's thesis, or the cutting plane algorithm (both are linked on the course webpage). You can implement discriminative training using the same features used in your generative parser, but to get significant gains, you will probably need to add some interesting features. You can do this easily for the lexicon by passing a more interesting implementation of `LexiconFeaturizer` to `Lexicon`.

Note that when you switch to discriminative training, you will likely encounter significant headaches with unary rules. Because discriminative models allow the weight for a unary rule to be positive, it is possible for trees with an infinite number of unaries to have infinite score (which will cause `UnaryClosure` to throw exceptions). Additionally, if you implement an agenda-based parser, positive weights mean that even uniform cost search will no longer be optimal because a zero heuristic is no longer admissible/consistent. A standard way to cope with these problems is to only permit 0, 1, or 2 unaries to be applied over a span, but other options are possible, for example, you could just force unary rules to always have a strictly negative weight.

Extensions: You should implement some kind of additional exploration in `AwesomeParserFactory`. One choice of extension is to focus on the grammar, using better annotation / refinement techniques like horizontal and vertical markovization to improve the accuracy of your parser. If you choose to focus on this aspect, however, you should do more exploration of various kinds of annotation. A

truly ambitious approach would be to explore automatic state-splitting.

Another direction is to focus on efficiency of the parser. You could, for example, implement a pruning technique like A* parsing or coarse-to-fine pruning and compare it to an exhaustive approach.

You could also experiment with adding more interesting features to non-terminal productions. This will require you to modify/implement your own `Grammar` class, but should not require too much additional work.

To summarize our expectations in this assignment, we expect that the bulk of your time will go into understanding the setup and then building a basic PCFG parser (agenda-driven or CKY). Once that is working, we expect you to try out a 2nd-order / 2nd-order grammar, which should take little additional work; this grammar should work much better than your original grammar. Then, you should implement some kind of discriminative training and also some interesting lexical features. You should also report at least some errors (with examples) that you parser seems to make often, though you need not do a detailed data analysis (unless you want to focus on that). Finally, we expect you to do some other kind of extension that you find interesting. We do not want this last part of the assignment to be something that will double your time on this assignment. However, not all of the extensions we mentioned will be possible unless you do decide to invest substantial additional time.

Grading: We will check that your generative parser with `-maxTrainLength` and `-maxTestLength` set to 40 gets an F1 of at least 80.0, but as always, higher scores will improve your grade. The speed of your parser will also influence your grade. We will check that your generative parser gets an F1 of at least 85.0 with `-maxTrainLength` and `-maxTestLength` set to 15, and that your discriminative parser improves on that by at least 0.5 F1. Your write-up should describe how you binarized/markovized your training data, what discriminative training method you used, and which additional features you used in your discriminative model. Your extension(s) will be evaluated primarily on your write-up. As always, the best submissions will be those that perform thoughtful error analysis and/or describe interesting extensions to the basic parsers in this assignment.

Coding Tips: Whenever you run the java VM, you should invoke it with as much memory as you need (and in server mode for JVMs which don't default to sever):

```
java -server -mx500m package.ClassName
```

If your parser is running very slowly, run the VM with the -Xprof command line option. This will result in a flat profile being output after your process completes. If you see that your program is spending a lot of time in hash map operations or `hashcode` / `equals` methods, you might be able to speed up your computation substantially by backing your sets, maps, and counters with `IdentityHashMap`s instead of `HashMap`s. This change requires the use of something like a `util.Interner` for canonicalization.

Submission: As usual, submit your jar to the online system (linked on the assignment page). Please check that your code runs for all three parsers with `-sanityCheck` enabled.