Treebank Parsing

Programming Assignment #2
Due: Thursday, October 22nd at 11:59pm

IMPORTANT NOTE #1 : You may complete this assignment individually or in pairs. *We strongly encourage collaboration*. Your submission must include a statement describing the contributions of each collaborator. See the collaboration policy on the course website: `http://web.stanford.edu/class/cs224n/grading.shtml`

IMPORTANT NOTE #2 : Please read through this whole document before starting on the assignment. We are eager to answer your questions, but please make sure that your question about section 2.3 isn't answered in section 2.4. You should also read the comments and Javadocs in the starter code files.

IMPORTANT NOTE #3 : If you have never logged into Farmshare, then please make sure you can well before the assignment deadline: `http://farmshare.stanford.edu/`

1 Introduction

This assignment looks at parsing. You will implement a parsing algorithm for a broad coverage statistical treebank parser and test your algorithm on the WSJ section of the Penn Treebank. We have included most of the support code to test your parsing algorithm.

2 Setup

We've put the Java starter code for this assignment in the directory `/afs/ir/class/cs224n/cs224n-pa/pa2/java`.

Login to Farmshare and copy the Java starter code to your workspace. Then build it using ant:

```
cd
mkdir -p cs224n/pa2
cd cs224n/pa2
cp -R /afs/ir/class/cs224n/cs224n-pa/pa2/java .
cd java
ant
```

The data for this assignment is located in `/afs/ir/class/cs224n/data/pa2/`. To avoid exhausting your AFS quota, you should not copy the data to your workspace.

We provide a testing harness for your PCFG parser. Make sure you can run the main method of the `PCFGParserTester` class. You can check this by running the following command from your java directory:

```
java -cp classes cs224n.assignment.PCFGParserTester
```

If your code compiled correctly, you should get a few trees, followed by two lines of output that look similar to (numbers may not always be the same):

```
[Current] P: 100.00 R: 100.00 F1:  100.00 EX: 100.00
[Average] P: 100.00 R: 100.00 F1:  100.00 EX: 100.00
```

We will explain what the various output fields mean in section 3.2.

3 Building A PCFG Parser

In this assignment, you will build a broad-coverage probabilistic parser.

3.1 Videos and References

We want you to focus on building a probabilistic generalized CKY parser[1], as is discussed in the short video segments on the CS224N website. Therefore, before starting this assignment, please be sure to first watch the required Week 3 & 4 videos on Probabilistic and Lexicalized Parsing.

The videos and lecture notes explain everything you need to implement the parser. For the part related to markovization, we highly recommend reading the following paper for more information:

- Dan Klein & Christopher D. Manning. *Accurate Unlexicalized Parsing*.
  `http://nlp.stanford.edu/~manning/papers/unlexicalized-parsing.pdf`

3.2 Testing Harness

At the beginning of the main method in `PCFGParserTester`, some training and test trees are read in. Subsequently, the training trees are used to construct a parser (as defined by the `-parser` argument; `BaselineParser` by default).

All parsers implement the `Parser` interface (which has two required methods: `getBestParse()` and `train()`). The parser is then used to predict trees for the sentences in the test set. For each test sentence the parse given by your parser is evaluated by comparing the constituency claims or brackets for each non- terminal node it generates with the constituency claims of the non-terminal nodes in the hand-parsed version[2]. From this, precision (P), recall (R), $F1$ score and EX score are calculated and displayed. The four metrics are calculated as follows:

$$P = \frac{100 \times \text{number of correctly guessed brackets}}{\text{number of guessed brackets}} \tag{1}$$

$$R = \frac{100 \times \text{number of correctly guessed brackets}}{\text{number of gold brackets}} \tag{2}$$

$$F_1 = \frac{2PR}{P+R} \tag{3}$$

$$EX = \frac{100 \times \text{ number of guessed trees that exactly match gold trees}}{\text{number of gold trees}} \tag{4}$$

Here's a list of command line arguments you can pass in to `PCFGParserTester`:

`-path`	The directory where the parsed data exists. By default this is `/afs/ir/class/cs224n/data/pa2`.
`-data`	Which data set to use. Pass in `miniTest` to train and test on the miniTest data set and pass `treebank` to train and test your parser on the WSJ section of the Penn Treebank dataset.
`-parser`	Defines which parser to run `PCFGParserTester` on. The default argument is `cs224n.assignment.BaselineParser` which runs the `BaselineParser`. Pass in the argument `cs224n.assignment.PCFGParser` to run the `PCFGParser` that you will be implementing.
`-maxLength`	Used to set the `MAX_LENGTH` variable described in section 3.4. By default this parameter is set to 20.

3.3 Code Overview

Before you dive into coding, take a moment to examine the `BaselineParser` class in order to understand how to use the other classes in the `cs224n.assignment` package. This baseline parser is really quite poor: it takes a sentence, tags each word with its most likely part-of-speech tag (i.e., acts as a unigram tagger), then looks for occurrences of the tag sequence in the training set. If it finds an exact match, it answers with the training parse of the matching training sentence. If no match is found, it constructs a right-branching tree, with labels for each node chosen independently, conditioned only on the length of the span of a node. If this sounds like a strange (and terrible) way to parse, it should, and you're going to provide a better solution.

You should familiarize yourself with these classes:

`cs224n.ling.Tree`	CFG tree structures (pretty-print with `cs224n.ling.Trees.PennTreeRenderer`)
`cs224n.assignment.Lexicon`	Pre-terminal productions and probabilities
`cs224n.assignment.Grammar`	CFG rules (both unary and binary) and accessors

`Tree` is a linguistic tree class that you will use in implementing your parser. `Lexicon` is a minimal lexicon, but it handles rare and unknown words adequately for the present purposes. You can use it to determine the pre-terminal productions for your parser; for example, $NN \rightarrow cat$. Formally, you would define it as the set L of items of the form $C^i \rightarrow x$, where C^i is a pre-terminal symbol and x is a terminal symbol. `Grammar` is a class you can use to learn PCFG rules from the training trees. The

[1] However, for extra credit, you can also additionally build any interesting and manageable (P)CFG parsing solution, such as a beam-decoded shift-reduce parser, and compare its performance with your CKY parser (see Section 5.2)

[2] More on evaluation metrics in the video on Constituency Parser Evaluation

grammar in this case is defined as the set R of items of the form $N^j \rightarrow \gamma$, where N^j is a non-terminal symbol and γ is a sequence of non-terminal or pre-terminal symbols. Since the training set is hand-parsed, learning a grammar is very easy. We simply set

$$\hat{P}(N^j \rightarrow \zeta) = \frac{c(N^j \rightarrow \zeta)}{\sum_\gamma c(N^j \rightarrow \gamma)} \tag{5}$$

where $\hat{P}(N^j \rightarrow \zeta)$ is the expected probability for a specific production involving N^j and ζ, and $c(N^j \rightarrow \zeta)$ is the count observed for any production involving N^j in the data set. Productions for the lexicon can be calculated in a similar way.

While you could consider smoothing rule rewrite probabilities, it is sufficient for this assignment to just work with unsmoothed MLE probabilities for rules. (Doing anything else makes things rather more complex and slow, since every rewrite will have a nonzero probability, so you should definitely get things working with an unsmoothed grammar before considering adding smoothing!) Note, however, that pre-terminal rewrites to words (productions in the lexicon) do need to be smoothed, and the `Lexicon` class implements a simple form of smoothing.

`UnaryRule` and `BinaryRule` are simply the classes the grammar uses to store these learned productions. They each bear the frequency estimated probabilities from the training set.

As discussed in the week 4 videos, most parsers require grammars in which the rules are at most binary branching. You can binarize and unbinarize trees using the `TreeAnnotations` class. The implementation we give you binarizes the trees in a way that doesn't generalize the grammar at all. You should run some trees through the binarization process to get an idea of what's going on. If you annotate/binarize the training trees, you should be able to construct a `Grammar` out of them using the constructor provided.

3.4 Your first job: implement the parser

Your first job is to implement a CKY parser in the `PCFGParser.java` file for the provided `Grammar`.

For the `miniTest` dataset your parser should match the given parse of the test sentence exactly. We strongly recommend that you get your parser working on the `miniTest` dataset before you attempt the treebank datasets. The `miniTest` data set consists of 3 training sentences and 1 test sentence from a toy grammar. There are just enough productions in the training set for the test sentence to have an ambiguity due to PP-attachment. There are unary, binary, and ternary grammar rules in the training sentences.

Once you've got this working you can move on to the `treebank` dataset.

Scan through a few of the training trees in the `treebank` dataset to get a sense of the 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 ternary-branching or longer. Currently, files 200 to 2199 of the Treebank are read in as training data, 2200 to 2299 are validation data, and 2300 to 2319 are test data (you can look in the data directory if you're curious about the native format of these files). At the moment, only the training and test set are used, but you are welcome to use the validation set too if you see a use for it. The static integer `MAX_LENGTH` determines the maximum length of sentences to test on (it does not affect the training set). You can lower `MAX_LENGTH` for preliminary experiments, but 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).

3.5 Your second job: add vertical markovization

Once you have a parser working on the treebanks, your next task is improve upon the supplied grammar by adding 2nd-order vertical markovization to the `TreeAnnotations` class in `TreeAnnotations.java`.[3] This means using parent annotation symbols like `NP^S` to indicate a subject noun phrase instead of just `NP`. You can test your new grammar on the `miniTest` data set if you want, though the results won't be very interesting, and you'll probably only see it working well on the `treebank`.

3.6 Expected results

With the default annotation scheme, you should be able to obtain an F_1 performance of around 60% on the Treebank dataset. This F_1 performance should increase by at least 3-4% following markovization and a score of 80% is achievable.

4 Performance Tips

Whenever you run the Java VM, you should invoke it with as much memory as you need with the `-Xmx` flag, specifying the number of gigabytes:

```
java -Xmx1g -cp ~/cs224n/pa2/java/classes cs224n.assignment.
```

[3] See section 3.1 for a reference on vertical markovization.

We strongly suggest using at least 1g to avoid running out of heap space. 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 hashmap, hash code, or equals methods, you might be able to speed up your computation substantially by backing your sets, maps, and counters with `IdentityHashMap` instead of `HashMap`. This requires the use of something like an `Interner` for canonicalization. Interning is documented extensively on the web, so Google + StackOverflow is your friend in case you need help implementing this.

5 Administrative Information

5.1 The Requirements

For the parsing portion, the following is expected:

- A PCFG CKY parser that can parse sentences of at least length 20 in a reasonable amount of time ($\sim$10 minutes for test set).
- An improved grammar with 2nd order vertical markovization.

5.2 Further Investigations for Extra Credit

If you have a parser which, given a test sentence, returns a most-likely parse in a reasonable amount of time, then you have done enough to get full credit. If you are interested in receiving up to 10% of extra-credit there are several ways in which you can embellish your parser:

- One choice is to focus on the grammar, using additional annotation and binarization techniques like horizontal and vertical markovization to improve the accuracy of your parser. The supplied grammar is equivalent to a 1st-order vertical process with an infinite-order horizontal process. You are required to implement 2nd-order vertical markovization, but you could also try 3rd-order vertical markovization or horizontal markovization, meaning forgetful binarization (symbols like `VP->...NP_PP` which omit details of the history, instead of `VP-> _VBD_RB_NP_PP` which record the entire history).
- Another choice is to focus on the efficiency of the parser. You could compare a beam method with exact methods like CKY or the agenda-driven method, or implement a pruning technique like an A* heuristic or a figure-of-merit and compare it to the basic agenda method (this may require substantial effort). Or you could study the trade-offs of pre-tagging the sentence before parsing, rather than letting the parser do the part-of-speech tagging. See section 3.1 for a reference on vertical markovization.
- A final choice is to investigate some other aspect of parsing that can be easily tested in your existing code. For example, you might investigate whether some compact subset of a grammar gives nearly the same accuracy. Or if you're interested in cognitive issues, you could study whether correct trees really do tend to have bounded stack depths in one direction or the other (and whether incorrect trees perhaps have different profiles). Or you could do an error analysis of the mistakes your parser makes.

5.3 Grading criteria

We will be grading based on both the F_1 score and the speed of your code. In particular we will run your code with 1Gb of RAM on Farmshare and record the execution time.

Important note about output format: Please do not leave any extraneous `println` statements, because we will be doing automated testing on your program outputs, and any extraneous statements will interfere with the grading script. Please also make sure that you do not change the output format for the P, R, $F1$ and EX metrics, as we will be matching them with a regular expression. Changing the output format might cause the automated grading script to fail. For the write-up, you should describe what you built, what choices you had to make, why you made the choices you did, how well they worked out, and what you might do to improve things further. There is a **hard limit of 4 pages + 1 page for extra credit**. *We may deduct 10% for each page that exceeds this limit.*

A detailed summary of what you should provide:

- A brief description of your implementation including any important design decisions you made.
- An evaluation of successes and failures of your parser. You don't have to do a detailed data analysis, but you should give a few examples to illustrate any errors your parser makes often and describe its performance in general.
- A discussion of further improvements you might make to deal with some of the observed errors.
- An explanation of any extensions you implemented and an analysis of whether or not they improved performance.

5.4 Submission Instructions

Please note:

- **We only accept electronic submissions**. *Do not submit a paper copy of the report*.
- If you are working in a group, then **only one member of the group needs to submit the assignment**.

You will submit your program code using a Unix script that we've prepared. To submit your program, first put your files in one directory on Farmshare. This should include all source code files, but should not include compiled class files. **Put your report PDF in the root of your submission directory**. Then submit:

```
cd /path/to/my/submission/directory
/afs/ir/class/cs224n/bin/submit
```

This script will (recursively) copy everything in *your* submission directory into the official submission directory for the class. If you need to re-submit your assignment, you can run:

```
/afs/ir/class/cs224n/bin/submit -replace
```

We will compile and run your program on Farmshare using `ant` and our standard `build.xml` to compile. If you did not complete the assignment on Farmshare, then please verify your code compiles and runs on it before submission. If there's anything special we need to know about compiling or running your program, please include a `README` file with your submission.