Advanced Statistical NLP: PCFGs and Parsing

Due 11:59pm PT, June 10, 2020

Collaboration Policy

You are allowed to discuss the assignment with other students and collaborate on developing algorithms at a high level. However, your writeup and all of the code you submit must be entirely your own.

Setup

You will need:

1. `assign_parsing.tar.gz`
2. `data_parsing.tar.gz`

`assign_parsing.tar.gz` contains a library jar, a build script, and a skeleton implementation for where your code will go.

Readings that you will find it useful to consult include Klein and Manning 2003 "Accurate Unlexicalized Parsing," Charniak et al. 2006 "Multilevel Coarse-to-fine PCFG Parsing," and Manning and Schuetze Chapter 11 Section 2.

Preliminaries

In this assignment, you will build an array-based CKY parser for English and, for extra credit, implement coarse-to-fine pruning. The main entry point for this assignment is `edu.berkeley.nlp.assignments.PCFGParserTester`. Run it with:

```
java -cp assign_parsing.jar:assign_parsing-submit.jar -server -mx2000m
  edu.berkeley.nlp.assignments.parsing.PCFGParserTester
  -path path/to/wsj -parserType GENERATIVE
```

You will run the two parsers that you build with `-parserType GENERATIVE` and `-parserType COARSE_TO_FINE`.

Data Currently, files 200 to 2199 of the Penn 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).

You can control the maximum length of training and testing sentences with `-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 against the gold trees using F1 over labeled spans (for the implementation, see `EnglishPennTreebankParseEvaluator`). Note that tags are *not* included in this evaluation; however, better tag prediction will of course help the parser do better on nonterminals that do matter.

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. 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.

Framework code You will be implementing two different parsing models. You will provide implementations of the `ParserFactory` interface in `GenerativeParserFactory` and `CoarseToFineParserFactory`. These factories construct (and train) a parser given a list of training trees. You should familiarize yourself with these basic classes:

`Tree`	CFG tree structures, (pretty-print with `Trees.PennTreeRenderer`)
`UnaryRule/BinaryRule`	CFG rules
`Grammar/UnaryClosure`	Builds and indexes the grammar
`SimpleLexicon`	Basic lexicon implementation

Note that very little of the code we provide you is required by the harness, so you can implement substitutes for these as you wish.

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 scores of all unary and binary rules in the treebank to their log relative frequency count. Nonterminals and preterminals in the grammar (but not words) are indexed according to an `Indexer`, which you can access with the `getLabelIndexer` method. `UnaryRule` and `BinaryRule` store rules as tuples of integers (indexed

nonterminals) associated with scores; rule indices are also arranged according to the integer representations of states.

`UnaryClosure` computes the reflexive, transitive closure of the unary subset of a grammar, and also maps closure rules to their best backing paths.[1]

The `SimpleLexicon` class provides basic functionality for scoring POS tags for words in the form of the `scoreTagging` method. It takes a treebank and collects some simple counts to use for computing probabilities. This lexicon is minimal, but handles rare and unknown words adequately for the present purposes.

Description

This project has two main components. The first component is the core assignment: building a generative parser. The second component is extra credit and involves making the parser more efficient by using coarse-to-fine pruning.

Core assignment: Build a generative parser

For your generative parser (`GenerativeParser`), you should:

1. Build an annotator/binarizer that takes raw trees, does parent/grandparent annotation and Markovization in order to return binary trees appropriate for learning a PCFG from
2. Implement CKY to compute Viterbi trees

The `TreeAnnotations` class provides a baseline implementation of binarization that doesn't generalize the n-ary grammar at all (convince yourself of this), so it will be slow and statistically inefficient. You will be implementing a more sophisticated binarization scheme on your own. Regardless, you can run some trees through the provided binarization process and look at the results to get an idea of what's going on. With binarized training trees, you should be able to construct a PCFG (represented by a `Grammar` and `SimpleLexicon`) out of them and run the whole pipeline.

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")))
```

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

[1] What this means is that in addition to rules $A \to B$ with probability p_1 and $B \to C$ with probability p_2, the `UnaryClosure` will expose a rule $A \to C$ with probability $p_1 p_2$ that maps back to the sequence ABC. This should make CKY easier to implement, as you can avoid having to build arbitrarily long chains of unaries (see Implementation Tips below for more about unaries).

history, instead of `@VP->_VBD_RB_NP_PP` which record the entire history), you should be able to get around 83-84 F_1 with `-maxTrainLength` and `-maxTestLength` set to 15.

Extra credit: Build coarse-to-fine pruning

For your coarse-to-fine parser (`CoarseToFineParser`), you should expand your CKY implementation to compute span marginals and pruning masks, then set up your parser to run in two passes with pruning.

The idea of coarse-to-fine is to use marginals from a coarse grammar (e.g. an X-bar grammar, which is $v = 1$, $h = 0$ according to Klein and Manning notation) to prune symbols in a fine grammar. For example, suppose the coarse model is very certain that NP never appeared over the span $(1, 3)$ based on the X-bar grammar; now in the fine grammar, you might rule out symbols such as $NP \wedge S$ and $NP \wedge VP$ (and Markovized variants) from occurring over that span. Ideally, your coarse-to-fine parser should spend the majority of its time parsing with the coarse (X-bar) grammar; the fine pass should take almost no time if you have your pruning thresholds set appropriately. You will want to experiment with this threshold to maximize the number of states you can prune while retaining good parsing accuracy on the second pass. Also note that the effect will be greater on longer sentences, so you may need to run on length 40 to see speedups.

To compute marginals, you will need to implement the inside-outside algorithm; see Manning and Schuetze Chapter 11 for reference. This algorithm allows you to compute for each span (i, j) and each symbol X the weighted fraction of trees containing X over that span.

Implementation Tips

Markovization and Tree Annotation Klein and Manning (2003) gives examples of what this looks like. Note that there are many choice points even for a simple $v = 2$, $h = 2$ implementation. Do you annotate a Markovized chain with the eventual final symbol? Where do you insert unaries? Do you do the same annotation to the preterminals as to the rest of the grammar? If your score is low, you are also free to implement other tricks from KM03 to try to improve performance.

Unaries Handling unary productions is one of the main tricky parts of implementing a textbook description of CKY (which usually assumes a grammar with unaries only to produce terminal symbols). There are many ways to handle unaries, but the easiest to parse with two charts, a "top" chart and a "bottom" chart. The CKY recurrence is used to apply binary rules to elements in the top chart to produce entries in the bottom chart, and then unaries are used to produce entries in the top chart. Unary rules of the form $A \rightarrow A$ with probability 1 are added for every symbol to allow for unaries to be discarded. This produces trees with alternating binary and unary rules: however, note that unary rules include identity rules ($A \rightarrow A$) which can be deleted as well as unaries produced by the `UnaryClosure` which can be mapped back to longer unary chains.

Note that you (probably) don't want to transform your training trees to look like alternating binaries and unaries; it would substantially change the rule scores you read off the treebank. Instead, this

method provides a way to implement CKY more simply if you're willing to rule out some possible trees (which is fine in practice: long unary chains are more often the product of pathological parser behavior than linguistically-attested phenomena).

Sums vs. maxes When implementing the Viterbi algorithm for inference in the generative parser, you should do it with an eye towards the coarse-to-fine step. In particular, marginals are computed by summing over trees (you can use `SloppyMath.logAdd` for summing numbers in log space) in the dynamic program whereas Viterbi trees are computed by maxing. You may wish to modularize your code such that maxing and summing are easily swappable.

Submission and Grading

Write-ups You should turn in a 2-3 page write-up as usual. You should clearly describe what implementation choices you made (how you binarized/Markovized the training data, how you implemented coarse-to-fine, etc.) along with reporting your performance on the various relevant metrics (F_1, parsing time, how these trade off when using coarse-to-fine, etc.) and providing error analysis.

Submission You will submit `assign_parsing-submit.jar` and a PDF of your writeup to Gradescope. Details will be posted to the course website. We will sanity-check with the following command:

```
java -cp assign_parsing.jar:assign_parsing-submit.jar -server -mx300m
  edu.berkeley.nlp.assignments.parsing.PCFGParserTester
  -path path/to/wsj -parserType GENERATIVE -sanityCheck
```

Grading A strong submission on this assignment will get around 80 F_1 on sentences of length 40 with `GenerativeParser`. Coarse-to-fine (worth extra credit) will then speed that up by a factor of 1.5-2. For reference, on trees of length 15 on the development set, our 2nd-order/2nd-order parser trains in about 10-15 seconds, and takes 17 seconds (on an i7 2GHz MacBook Pro) to decode for a score of 86.01 F_1. On length 40 on the development set, we take 664 seconds and get a score of 79.92 F_1. (This implementation is pretty optimized for accuracy but not as much for speed.)

Because coarse-to-fine parsing incurs additional overhead (requiring an outside pass), you may not be able to get large speedups if your grammar is small. If you don't see much in the way of speedups, try analyzing why that might be the case in terms of your grammar and the amount of work your parsing is doing. (It's worth noting that more complex grammars that obtain 90+ F_1 would be much more complex still and see larger improvements from coarse-to-fine.)

As with past projects, implementing additional interesting techniques and doing extra analysis will result in a higher score.