Natural Language Models and Interfaces

Project description (Part B)

May 3, 2013

In this assignment, you will build a statistical syntactic parser for English. The PCFG model will be estimated from a treebank (Penn Treebank WSJ) and then applied to new sentences using an efficient polynomial-time parsing algorithm (CKY). The performance of the parser will be evaluated by comparing its predictions to human annotations present in the treebank. You are provided with source code for a baseline (but not quite complete) parser which you will modify to make it fully functional. The project will consist of 2 stages, each with its own deadline (see the web-page or Blackboard for the dates). Make sure you can access the source and data files (on Blackboard).[1]

1 Code and Data

The main class for the assignment is: `nlp.assignments.parsing.PCFGParserTester`. The provided code supports reading, preprocessing, estimation of the grammar, basic parsing, post processing and evaluation.

Assuming that you built the code and the binaries are in the class path, you should be able to run the baseline parser:

```
java -server -Xmx1G nlp.assignments.parsing.PCFGParserTester
```

The JVM parameters make sure that enough memory can be allocated, as well as enable optimization which (depending on JVM you use) is likely to substantially speed up parsing and parameter estimation.

However, as the parser is not fully functional (namely because it does not support unary rules), it will throw an exception when parsing the first sentence. In order to understand how the parser works before you extend it to support unary rules you can use the toy dataset:

```
java nlp.assignments.parsing.PCFGParserTester -path toy-data -no-binarization
```

The dataset `toy-data` is including in the package downloadable from Blackboard. It contains two syntactically annotated sentences for training (in `toy_0200.mrg`) and one sentence for evaluation (`toy_2200.mrg`), all internal nodes in these trees have two children (except for preterminals, i.e. PoS tags). These examples are nonsensical (both linguistically and semantically) but included to help you to understand how the parser works before you implemented unary rules. The flag `-no-binarization` switches off grammar transformations and annotations (we will discuss them at the May 21 lecture), as the transformations will otherwise introduce unary rules. Do *not* use this flag unless playing with the toy dataset.

The workflow of the parser is detailed below. As always, you may also choose to step (in the debugging mode) through the code to understand how it works. Please also see comments in the code which point to places which need to be modified or added (at one of the assignment stages), as well as clarify some of less obvious bits.

Loading the data. The process starts with reading the data. By default the data is assumed to be located in the local subdirectory `data`, however a different path can be set using `-path` option. The

[1] The code is mostly based on the one kindly provided by Dan Klein and Slav Petrov. The data is licensed by LDC. Both the data and the code are not to be redistributed.

training data by default consists of sections 02–21 of the treebank, evaluation is performed on section 22. The final evaluation can also be performed on the standard test set, section 23 (if the `-test` option is set). To reduce running time, we will use only sentences of length at most 20 words (both for testing and training). All these parameters can be changed by editing `PCFGParserTester`. However, make sure that these parameters are preserved in the versions you submit at both stages.

Binarization and grammar estimation. When the data is loaded, the `BaselineCkyParser` object is created. `BaselineCkyParser` is one of the classes which you will modify in this assignment. Unless `-no-binarization` is set, before estimating the grammar, the trees are converted into the binary form (aka horizontal lossless Markovization which will be discussed in class). This preprocessing is accomplished by `BaselineTreeAnnotation`, this class you will modify at the substage 2.1 to support more effective forms of grammar annotation.

The resulting transformed trees are used to induce the probabilistic grammar (i.e. PCFG parameters). The PCFG parameters are computed and stored in two classes: preterminal productions (generation of words given part-of-speech tags) are handled by the class `Lexicon`, parameters of the remaining binary and unary rules are computed and stored in the class `Grammar`. Unlike `Grammar`, the class `Lexicon` is capable of computing the scores for unseen productions as needed to handle unseen words, and performs some basic forms of smoothing for rare words.

Parsing and evaluation. The method `testParser()` of `PCFGParserTester` deals with evaluating the parser performance. It calls the method `getBestParse(...)` of the parser and provides sentences (a list of its words) as an input. The included implementation of the CKY algorithm (see `BaselineCkyParser.getBestParse(...)`) supports only binary rules and consequently cannot parse sentences from the real dataset. This shortcoming you will fix at the substage 2.1 of the project. Note that before returning the parse tree to `PCFGParserTester` the parser converts it back to the original grammar using the `BaselineTreeAnnotation.unAnnotateTree(...)` call.

The parser output is evaluated against the gold standard (i.e. the parse trees provided by human experts). The scoring measures (bracketing precious, recall, their harmonic mean F1 and exact match) will be discussed in class but for simplicity you can assume that the third score (F1) is the most important one. See `EnglishPennTreebankParseEvaluator` for details. Note that PoS tags are *not* included in this evaluation.

After parsing each sentence, the evaluation scores for this sentence are printed (marked as *[Current]*) along the evaluation scores for all the processed sentences (*Average*). After parsing the last sentence, the final results are reported. If you prefer to have less information printed on the screen, use the flag `-quiet`.

2 Computing the Probability of a Tree (stage 1)

For the stage 1, you will spend most time familiarizing with the key classes, namely, `Grammar`, `Lexicon`, `BinaryTree` and `UnaryTree`, as actual implementation would require very little coding.

The task is to compute the probability of a given tree according to the probabilistic grammar induced from the treebank. In order to do this, you will need to implement the method `double getLogScore(Tree<String> tree)` in `BaselineCKYParser`. It will return the logarithm[2] of the tree probability according to the grammar (as represented in the objects `grammar` and `lexicon`). Roughly, you will need to traverse the tree from the root, find out which rules are used (either of the type `BinaryRule` or of the type `UnaryRule`) and compute the score by summing the logarithms of the rule probabilities (recall $\log xy = \log x + \log y$). Some of the rules (other than preterminal rules) may be unseen in the training date, it is natural to assume that their probability is 0 (i.e. the log probability is `Double.NEGATIVE_INFINITY`).

In order to test how well your computation works, use the command line

```
java -server -Xmx1G nlp.assignments.parsing.PCFGParserTester -scoring-mode
```

[2]The reason for using logarithms is that the tree probabilities would be very small and computers are not very good at representing very small numbers (google *'arithmetic underflow'*).

The log-probabilities for the first 10 trees in the validation set will be printed. Do not be surprised that some of them (3?) will be reported as $-Infinity$ even if you implemented everything correctly.

When submitting your code (and a short description, at most 1 page), please make sure that you have not changed `PCFGParserTester`. This stage will be graded as passed / not passed.

3 Adding Support of Unary Rules in the CKY algorithm (substage 2.1)

The CKY algorithm implementation in `getBestParse(...)` does not support unary rules. We will discuss this extension in class on May 17. In order to successfully pass the stage 2.1, you will need to understand how the baseline CKY implementations works (including the `Chart` and `EdgeInfo` subclasses) and extend it to support the unary rules (unary preterminal rules are already supported). The provided class `UnaryClosure` should be a big help to you, though you are free to implement it from scratch if preferred. Please see the comments in the code to understand better which classes need to be changed. There will be 3 main modifications: testing if (for some C') an unary rule $C \rightarrow C'$ (or really a chain of unary rules $C \rightarrow \ldots \rightarrow C'$) can be applied for each edge (i, j, C) to improve the score `chart.get(i,j,C)`; modify back pointers to support unary rules and modify the method `traverseBackPointersHelper(...)` which follows back pointers when creating a tree.

After you are done, you should be able to run the parser on the real data. If your implementation is correct (or nearly correct), the overall parsing $F1$ should be around 76-77% in our set-up (and exact match, i.e. the percentage of trees predicted absolutely correctly, will be around 23%). Also the parsing time should not exceed a couple of seconds per sentence.[3]

4 Extending the Treebank PCFG (substage 2.2)

At this stage you will need to extend the treebank grammar. The baseline tree annotation (i.e. lossless Markovization) is implemented in `BaselineTreeAnnotations`. In this assignment you will need to modify the method `annotateTree(...)`. Minimally you should implement horizontal and vertical Markovization discussed in class (on May 21). Try at least the horizontal order $h = 2$ and the vertical order $v = 2$ (the baseline tree annotation can be regarded as $h = \infty$ and $v = 1$). The results in this mode should be around 81% F1 and 29% exact match.[4]

However, we would be happy to see (and it will be reflected in grading) if you (additionally!) come up with your own ideas and implement alternative (and hopefully more accurate) methods. In this case, you may want to have a look into Klein and Manning [1] for inspiration. The submitted version of the code should have the best method, in terms of F1, enabled (i.e. either the best method you came up with, or $h = 2, v = 2$ Markovization if it still works better).

Unannotation (i.e. debinarization and stripping of extra information) should be performed automatically (you do not need to touch `unAnnotateTree(...)`) as long as you stick to the convention used in the class. For example, labels of the format $a\hat{}b$ or a-b will be converted to a, nodes having the format $@x$ will be assumed to be a result of binarization and spliced out. Please also see comments in the code or / and debug through the code of lossless Markovization.

Rerun the experiment done at stage 1 while using the new annotation strategy: have the probabilities changed with the new annotation method? If they generally increased, can you speculate why?

The code and the final paper should be submitted simultaneously for substages 2.1 and 2.2. Please make sure that the following command line results in using the baseline grammar transformation (i.e. lossess Markovization; substage 2.1):

[3] Our implementation does not use any kind of optimization to make it easy to understand and to follow the pseudocode presented on the slides as closely as possible. A reasonably optimized parser is capable of processing several dozens of sentences (≤ 20 words) per second with a grammar of that size.

[4] The boost in performance would be greater when using the full testing and training set (rather than ≤ 20): roughly from 72% to 78% F1 with $h = 2, v = 2$ Markovization.

```
java -server -Xmx1G nlp.assignments.parsing.PCFGParserTester -lossless-binarization
```

Whereas the annotation strategy developed in substage 2.2 is used when running:

```
java -server -Xmx1G nlp.assignments.parsing.PCFGParserTester -grammar-annotation
```

You will need to edit (in a trivial way) `PCFGParserTester.main(...)` to support this.

The final paper, summarizing stages 1, 2.1 and 2.2 submitted along with the last version of the code, should discuss your experiments (e.g., annotation strategies you tried) and findings (at most 4 A4 pages total). Please follow exactly the same procedure to submit your assignments (code and papers) and use the same format as was requested for the Part A of the class. The only difference is that your code should support the command line format specified above.

References

[1] D. Klein and C. D. Manning, *Accurate Unlexicalized Parsing*. Proc. of Association for Computational Linguistics, 2003. http://acl.ldc.upenn.edu/P/P03/P03-1054.pdf