Elements of Language Processing and Learning

Khalil Sima'an and Gideon Wenniger
Institute for Logic, Language and Computation (ILLC)
University of Amsterdam, Amsterdam, The Netherlands
k dot simaan the-at-sign uva dot nl

October 31, 2011

Project 1: Statistical Parsing The project concerns building a prototype statistical parser. We will not dwell on issues of efficient representation or faster parsing, but we should keep an eye that we use suitable packages for the task. The tasks have been simplified a lot for your convenience. We do not take care of unknown words, we do not treat lexicalized parsers and we do not take care of smoothing production probabilities. This is a bit of a "crippled" parser and it is up to you to make it better at the final step.

Data At your disposal you find a downloadable treebank (Penn WSJ). It consists of three parts:

- Training Data: on this part you train your parser.
- Test sentences: the sentences that your parser will be tested on.
- Test Gold Standard trees: this part you keep untouched and use only at the end for evaluating the output of your parser. This file contains the correct tree for every sentence in the Test sentences.

This Training Data part of the treebank has been prepared to make life easy for you: the trees are in binary form (the node labeled with @ have been added by us) and there are no unary productions. The sentence length in the Test Data is limited to 15 words or less (in the test set).

The project consists of three steps:

Step 1: Extract PCFG from Treebank Extract all Context-Free Productions from the Training Data trees. Store these productions in a *data structure* that allows fast search for productions with a certain right-hand side and left-hand side. You will use this data structure for the CYK parser (next step), so try to take that into consideration. You will also store with every production $A \rightarrow \alpha$ also its probability $P(A \rightarrow \alpha \mid A)$ in a data structure. Estimate the probabilities using relative frequency in the treebank (i.e., Maximum-Likelihood estimate of the treebank when assuming it is generated by this PCFG).

Tasks:

- Program this and test it for correctness.
- Document the choices you make (data structure) in a two page (11pt font) document (PDF only) where you start out with writing about what PCFGs are, how the probabilities of productions, derivations, trees and sentences are defined under PCFG models. And continue with the implementation choices and then report some empirical findings as in the next point.
- In the document, give examples of four words that are each (separately) syntactically ambiguous and list the probabilities of the productions where these words are found. For each X category in the set $\{VP, S, NP, SBAR, PP\}$ list four most likely productions $X \rightarrow \alpha$ with their probabilities in a table.

Deliver: The work in this step is reported and documented as part of Step 2, next.

Step 2: Implement CYK/CKY Employ the data-structure representing the treebank PCFG you extracted from the first step and now program a CYK (or CKY as some call it) algorithm for parsing under this grammar with the following specifications:

Given: Treebank PCFG G (and the data-structure representing it).

Input: On a single line a sentence U consisting of a finite sequence of words as in usual text.[1]

Output: All productions $TOP \rightarrow XP$ that cover the whole sentence U (entry $\langle 0, n \rangle$, where n is number of words in U) according to your CYK parser. In the process, the CYK parser builds a *parse-forest* $CYK(U)$ for input U.

[1] You might want to have a look at the Text Data to get an impression of how sentences look like in your input (including the stop symbols in there - which you can ignore by adding your own at the end of every input line).

Tasks:

- Program the CYK parsing algorithm that loads the grammar G when it starts, reads a single line (sentence U) from the stdin or a file and then builds the parse-forest for sentence U under G. [2]
- Parse with your own parser the whole Test Data, one sentence at a time. And then deliver a file that contains for every Test Data sentence in turn (i.e., in the order of the Test Data) the Output of your parser as specified above.
- Write a document of one to max. two pages (PDF, 11pt font) where you describe at the abstract level and at pseudo code the CYK algorithm. Comment as you wish on your implementation detail and report any interesting experience you have had.

Deliver (see submission guidelines on blackboard):

- Program code with clear README file.
- Report regarding Step 1 and 2 in PDF.

Step 3: Implement Viterbi (and possibly Inside) You now have the grammar G data structure, a CYK algorithm implementation that builds a parse-forest $CYK(U)$ for every input sentence U. In this step you will program a statistical version with Viterbi disambiguation and evaluate your parser against the Test trees.

Tasks:

- Using the CYK implementation, implement now a recursive procedure that calculates the Viterbi parse and its probability for sentence U.
- Extract the Viterbi parse from $CYK(U)$ and print it out in the same format as the Penn WSJ format. While printing a tree you should remove the nodes labeled with so you can compare to the Test trees (which are in original Penn WSJ format, unbinarized).
- Extra (not demanded but would improve your results): think up an easy solution for dealing with words in the Test sentences that do not appear in the Training set (i.e., unknown words). One simple solution is to add every

[2] When filling the chart entries keep pointers that would allow you to extract a parse-tree from the parse-forest. You will need these pointers for the next step, when you will extract the most likely parse under the PCFG G.

unknown word from the test set with a few POS tags as extra lexical productions that extend PCFG G into one that can deal with all words in the Test sentences. Which POS tags will you select for a word? what are the probabilities of the new lexical productions? Or maybe you want to look up an easy solution in Michael Collins' paper (see reading list).

- Use the program evalC for evaluating your Viterbi parses against the Test trees and report Labeled Precision, Labeled Recall and F1 score with a few more interesting insights into where the parser is doing well and where it is giving worse results. EvalC is downloadable from
  http://staff.science.uva.nl/~fsangati/evalC_25_05_10.zip

Deliver (see submission guidelines on blackboard):

- A report (max two pages, 11pt font, PDF only) that documents the mathematical description of Viterbi and Inside and the pseudo-code. Some description of your program choices and the results of the parsing, some analysis of the outcome of your parser and ideas on how to improve it.
- Program code with clear README file (see submission guidelines on blackboard).