**Homework Assignment #6**

I implemented a probabilistic CYK parser, trained and tested on the text data provided on Moodle. In the learning phase, the parser merges words that occur only once in the training set into <UNK> to estimate the probability of unseen words and deletes non-terminal rules that occur only once in the training set to improve the accuracy of the parser. It then prints out the resulting parse tree to a file. Accuracy measurement was done using EVALB.

A dictionary of production rule probabilities was extracted using a modified version of the helper code provided on Moodle. This modified rule extractor merges all words occurring once into <UNK> to estimate the probability of unseen words and deletes selected non-terminal rules to prevent the parser from giving a false probability of 1.0 to rules occurring only once in the training data, such as the rule SQ → NP VP. Finally, it converts the probabilities obtained into log probabilities before printing them out to a file.

The parser reads the file containing the production rule probabilities and stores it to the variable `grammar`. Parsing is done the same way it is done in non-probabilistic CYK, except that the program has to keep adding the log probabilities and store this information for each possible constituent. The program also keeps a table of backpointers to trace back and print the tree after parsing is completed. However, instead of looking for the 'S', it looks for the highest probability obtained in the last column of the first row and use the information stored in the backpointer table to trace back from there. If the parser fails to parse a sentence (i.e., when it ends up with a blank cell in the first column of the first row), None is returned. For example, the parser cannot recognize that the word "That" in the sentence "That is a bit unfortunate" is an NP, since the word never occurs as an NP in the training data. When attempting to parse this sentence, the parser got stuck and was unable to finish the parse, leaving the first-row-last-column cell blank. Since there was no tree that could be drawn from an incomplete parse, None was returned. Successful parse trees are stored in a file which is then compared against the gold standard and evaluated using EVALB. The evaluation result is summarized below:

| | |
|---|---|
| Number of sentence | = 98 |
| Bracketing Recall | = 86.18 |
| Bracketing Precision | = 86.18 |
| Bracketing FMeasure | = 86.18 |
| Tagging accuracy | = 75.14 |

The most challenging part of the assignment (that took the longest time to figure out) was to obtain the correct parse for NP VP. Because SQ occurs only once in the training data with the rule SQ →NP VP, the probability of NP VP producing SQ is 1.0, higher than any other possible NP VP production rules. This is incorrect, because most NP VP constituents produce S. To

handle this, I modified the rule extractor to delete all non-terminal rules occurring only once. Another problem I encountered, that was left unsolved because I ran out of time, was to handle an unsuccessful parse (i.e., first-row-first-column cell left blank).

If I could build on what I've done so far, I would try using other methods to handle unseen words, such as parsing the POS tags after running a POS tagger first, instead of parsing the words, or employ other smoothing methods. My parser is very slow whenever an unseen word is encountered because it stores all possible tags (left-hand sides) that have an <UNK> right-hand side to the dynamic programming table. This means there are so many comparisons that need to be done to fill in each cell. Therefore, a faster approach to handle unseen words is necessary. Additionally, I think it would be interesting to further investigate how to handle parsing failures where the first-row-last-column cell is left blank.