Homework #2: CMPT-825

Anoop Sarkar – anoop@cs.sfu.ca

For the programming questions use the `makefile` provided to you.

(1) **Beam Search Thresholding**

This question will introduce you to the basics of statistical parsing. The task will be to speed up an existing statistical parser written in Python in a sequence of steps listed below.

A recurring theme in statistical parsing is the need to limit the search space of the parser. Many productions that a hand-constructed context-free grammar (CFG) would rule out as impossible, a learned probabilistic context-free grammar (PCFG) might permit with low but non-zero probability. Techniques to limit the search space might still guarantee that the optimal parse is found (such as A* search). Or they might do more aggressive pruning for even more speed, but sometimes fail to find the most probable parse. The Beam Search Thresholding you will implement is an example of this second, more aggressive pruning technique.

You will be taking an existing probabilistic context-free CKY parser and adding beam search thresholding to improve its running time. The CKY algorithm without pruning is exhaustive, guaranteed to find every possible parse of a sentence. CKY for PCFGs (also called Viterbi for PCFGs) are guaranteed to find the parse with highest probability according to the grammar.

Beam search thresholding is a particular form of pruning which ignores items stored in the parser chart that have a very low probability, relative to other items in the same cell. Pruning these items is like making a bet that these items are so unlikely, they will not be in the most probable parse, even if they possibly might combine with other high-probability items for larger spans. The ratio in probabilities from the most probable item in a cell to the lowest probability permitted for a “competing” item is the Beam Width, and this parameter can be tuned to trade off speed (a narrower beam can prune more items) vs. accuracy (pruning more items risks pruning the most probable parse).

The hand-annotated (also called gold) reference parses, and the output of the provided parser, use the standard nested parentheses to represent the parse structure. The evaluation of the quality of parses uses the `evalb` tool written by Satoshi Sekine and Michael Collins, which reports the precision and recall of the brackets (the parse tree nodes and their spans) relative to the reference parse.
Here’s an example of what the evaluation is:

```
candidate: (S (A (P this) (Q is)) (A (R a) (T test)))
gold:      (S (A (P this)) (B (Q is) (A (R a) (T test))))
```

In order to evaluate this, we list all the labeled brackets with their spans, skipping the brackets of the form `(A a)`, i.e. those brackets that have a span of 1 (note that evaluation of spans of length 1 would be equivalent to evaluating the part of speech tagging accuracy):

Candidate	Gold
(0,4,S)	(0,4,S)
(0,2,A)	(0,1,A)
(2,4,A)	(1,4,B)
	(2,4,A)

Precision is defined as $\frac{\#correct}{\#proposed} = \frac{2}{3}$ and Recall as $\frac{\#correct}{\#in\ gold} = \frac{2}{4}$.

The F-score (or balanced F-score) is defined as harmonic mean of Precision and Recall, defined as:

$$F = 2 \cdot \frac{P \cdot R}{P + R}$$

Use the `evalb` program to check the values of Precision and Recall for the above example. It is installed in the directory `EVALB` inside the ~`anoop/cmpt825/sw` directory.

Here are the data files you will be using for the experiments:

- Grammar: `atis3.gr`
- Part of speech tags to be used for unseen words: `atis3.unseen`
- Sentences to be parsed: `atis3.test.sents`
- "Gold" reference parses for the input sentences: `atis3.gold`
- Pretty printing of parse trees: `indentrees.pl`

Notes on the grammar: The grammar was extracted from the Penn Treebank of Air Travel Information System (ATIS) articles with no smoothing of the count statistics. The grammar was converted into a form that is (almost) in Chomsky Normal Form. The implementation has some additional code to handle unary rules (rules of the form: $\mathtt{A} \rightarrow \mathtt{B}$), since the grammar used to annotate the treebank include many nodes with a single non-terminal as a (Right-Hand Side) RHS.

Optionally copy the above files to your directory. Copy the Python program `ckyparse-nopruning.py` as the file `answer/ckyparse.py`. You will modify this Python program to complete the homework. `ckyparse.py` allows you to set many options on the command line. The main ones are:

`-v`	goes into verbose mode
`-h`	prints a message on how to run the parser and quits
`-g`	provides a grammar file to the parser (use atis3.gr)
`-s`	provides a start symbol (TOP by default)
`-u`	file contains tags used for unseen words (use atis3.unseen)

a. Run the provided parser on the input file:

```
python ckyparse.py -g atis3.gr -u atis3.unseen < atis3.test.sents > out
```

The code provided to you does the job of parsing and recovering the most likely parse tree using the CKY algorithm for probabilistic context-free grammars. However, it runs without any pruning and it is relatively slow even on these short ATIS sentences. The above command creates the output file `out` containing the parses for the input `atis3.test.sents`. Compare the output parses to the gold reference parses in `atis3.gold` using the program `indentrees.pl`:

```
cat out | perl indentrees.pl | more
cat atis3.gold | perl indentrees.pl | more
```

Check the accuracy of the parser output using `evalb`:

```
evalb -p COLLINS.prm atis3.gold out
```

The Bracketing Recall and Precision figures reported by `evalb` are the best the CKY parser can do with this training data. We will try to maintain this accuracy while trying to speed up the time taken by the parser.

b. Extend the function `prune` in the `CkyParse` class so that after the parser outputs its best parse for a sentence, you can output the mean, max, and min number of items per chart cell for this sentence. Collect these statistics by modifying the `prune` function. Print these values to `sys.stderr` using the format:

```
# Items/cell mean: X.X max: N min: N Total items: N cells: N
```

c. Extend the function `prune` further to also compute the maximum and minimum log-probabilities of the items in each cell, and the log-ratio of max:min probabilities in that cell. Output another line to stderr after the Items/cell output in the following format:

```
# Log max:min probability log-ratios mean: X.X max: X.X min: X.X
```

Collect these statistics by modifying the `prune` function. (Note: each non-empty cell has a single max:min ratio, and your output should show the statistics for this value across all non-empty cells.)

d. Modify the function `prune` in the file `ckyparse.py`. Remove all items in each cell i, j whose probability is less than that of the most probable item in the same cell multiplied by β. The value of β is stored in the variable `beam` which can be assigned the value log(*value*) using the command line option `-b value` to the parser (the `-p` option enables pruning):

```
python ckyparse.py -p -b 0.0001
```

For example, let us assume that the value of the beam, $\beta = 0.0001$. Let us assume that the chart entry for the span `2,4` already contains the following non-terminals: `NP, VP, PP`. The list of non-terminals for each span i, j is stored in `chart[(i,j)]`. The log probability that a given non-terminal X derives the input from i to j is obtained by calling the function `chartGetLogProb(i, j, X)`.

In this example, let us take $P(\texttt{NP}, 2, 4) = 0.12$, $P(\texttt{VP}, 2, 4) = 0.03$, $P(\texttt{PP}, 2, 4) = 0.000011$. Beam thresholding would imply that we prune away $P(\texttt{PP}, 2, 4)$ since it is less than the maximum probability for the same span times the beam value: $P(\texttt{NP}, 2, 4) \times \beta$.

$P(\texttt{VP}, 2, 4)$ is not pruned since it sneaks by under the beam threshold.

Important: Remember that unlike the above example the values in the chart and the variable `beam` are stored as log probabilities.

Measure accuracy (using `evalb` as above) using beam thresholding with several different beam widths. Include the beam values 10^{-4} and 10^{-5}. You will observe that the F-score drops by a small amount but parsing times are reduced.

It's important to use the `-p` option which enables pruning in the `prune` function and to provide the beam value using the `-b` option. The source code that takes the values from the command line and assigns them to the right variables in the parser is provided to you already as part of `ckyparse.py`. You can simply use the variable `self.beam` in your code and assume that it has the right value taken from the command line.

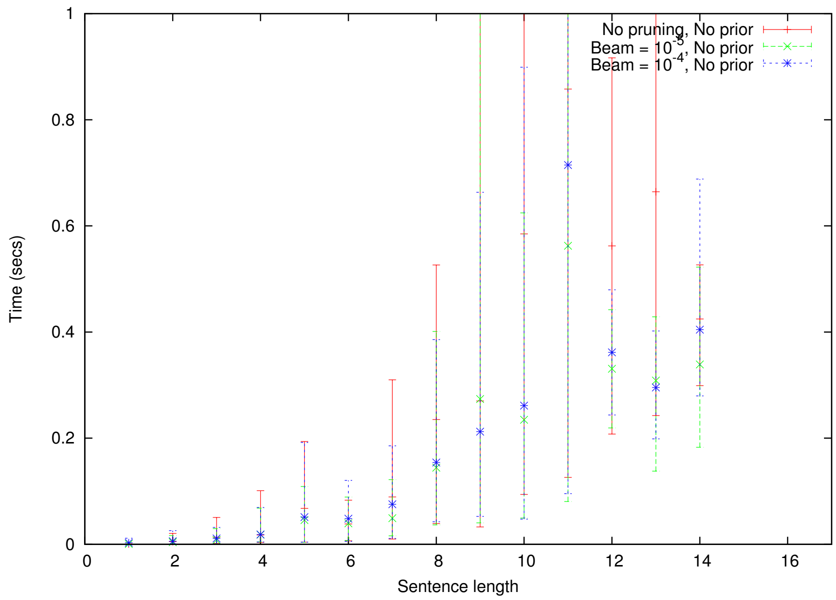
e. A problem with pruning using beam thresholding as we have done in the previous step is that some non-terminals can get the highest probability for a span, but when combined with other non-terminals in the right-hand side of a CFG rule, it might lead to a very improbable derivation. For example, $P(\texttt{FRAG}, 2, 4)$ might be very high but the rule $\texttt{VP} \rightarrow \texttt{FRAG PP}$ might be very improbable leading to search errors.

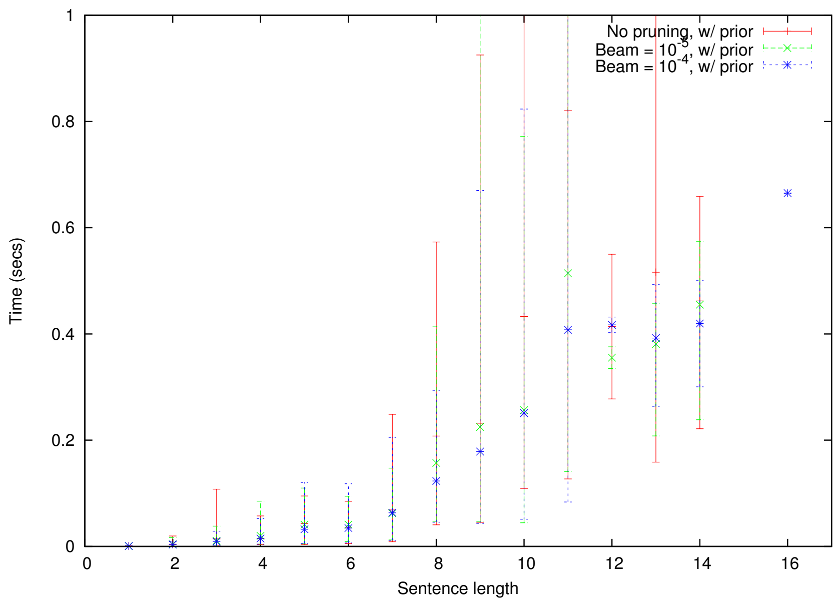
A solution to this problem is to use a rough estimate as to how likely a subtree will lead to full parse tree. This rough estimate can be as simple as the probability of a non-terminal independent of any context. This is the *prior* probability of a non-terminal. In the example above $P_{prior}(\texttt{FRAG})$ would be multiplied with the probability $P(\texttt{FRAG}, 2, 4)$ before we do beam thresholding as in the previous step.

Multiply the prior probability for each non-terminal in the pruning process. The `getPrior` method in the `pcfg` class returns the prior probability for a non-terminal. Within the `prune` function the prior probability for non-terminal X can be obtained by calling `self.gram.getPrior(X)`.

Provide the accuracy of the parsing with prior probabilities combined with beam values of 10^{-4} and 10^{-5}.

f. Plot the change in parsing time, comparing the parser without pruning and the parser with pruning at the various beam widths. Provide this graph which should show that your pruning strategy succeeds in speeding up parse times as you tighten the beam threshold value. Plot the output with and without the use of priors to check that adding the prior does not significantly slow the parser. Your output graphs should look similar to the following graphs (notice the use of error bars to show the variation in parsing times for sentences of the same length):


No pruning, No prior
Beam = 10^{-5}, No prior
Beam = 10^{-4}, No prior
Time (secs)
Sentence length
0
0.2
0.4
0.6
0.8
1
0
2
4
6
8
10
12
14
16


No pruning, w/ prior
Beam = 10^{-5}, w/ prior
Beam = 10^{-4}, w/ prior
Time (secs)
Sentence length
0
0.2
0.4
0.6
0.8
1
0
2
4
6
8
10
12
14
16