Ling 571 HW3

Michael Roylance, Olga Whelan

Structure of collaboration:

We completed homework 3 by doing pair programming. Over the first two days of work on the assignment, we worked independently and prepared each our own working versions of grammar induction, tentative data structures and outline of the pcky implementation. After that, in the course of five meetings we continued to work on the programme together.

First, we compared our implementations of the first stages of the project, discussed them and chose which one to pursue. Then, during our meetings we would programme together for 2-3 hours, while constantly discussing new code and our general strategy.

In the end of each meeting, we summarized what was achieved, set time and made plans for the next meeting, discussing the possible routes to pursue and looking for the material we should read up on.

For collaboration we used Skype, join.me and github.com.

Approach:

The programme includes three source files: ***main.py, induceGrammar.py, pcky.py***.

main.py: the parse trees are read line by line from the first input file. With the help of *nltk* and methods of class *induceGrammar,* pcfg is assembled and output into file *train.pcfg*. Example sentences from the second input file are parsed one by one and the best parse tree is returned to *main*.

induceGrammar.py: receives grammar productions one by one from main and builds three dictionaries (*fillDicts* - full grammar with frequencies of productions; *getLHSAndTerminals* – terminals and non-terminals). *getProbs* finds probabilities for each production (count/total) with the help of *probFromFreq* which calculates the total by summing up values for all keys. *buildPCFG* formats the grammar for the output. *getDS* grabs the filled data structures that will be needed later.

pcky.py: implements the algorithm and returns the best parse. Method *putDS* processes the data structures received as an argument of the class from the instance of *induceGrammar.py* and creates an additional list that will be helpful during the improvement stage. All the main work is done in method *runCKY*. Compared to hw2, now each cell stores not only a tree, but also its probability: *(tree, prob)*. At the end of runCKY, the recursive method *parseTree* takes care of the peculiar format of the output of the parse (complicated by probabilities) and converts it into format readable by *evalb*. For each input sentence, *fullString* contains the final 'clean' parse that is sent back to the calling class.

In this assignment we used *nltk* twice:
1. to induce the grammar from the cnf parse trees *(nltk.Tree.parse())*;

2. to retrieve parses *(nltk.tree)*. As we fill out the matrix, we build trees and store them in each cell.

Improvement:
Originally we did not get parses for 8 sentences. We solved the problem for 6 of those cases, which contained unknown words. This was achieved by creating additional initial trees on the superdiagonal of the matrix with all possible tags for the unknown words (i.e. a tree that assumes it's a NN, a VB, a JJ and so on). We supplied these trees with a 0.0001 probability, which is just a random value. This approach slows down our performance, but completely eliminates the problem of unknown words.

We also attempted to implement parent annotations in pcky.py, we left our code in between lines 57-72 commented out. Our initial try at it altered only the right hand side with the left hand side appended, but this was ultimately incorrect and resulted in shockingly low accuracies. During later discussion, we think we arrived at the correct approach which would have involved a recursive examination of the parsed tree at the beginning when we induce the grammar. However, we were running out of time and decided not to pursue this further for this assignment.

Baseline

Number of Sentence	**Number of Error**	**Number of Skip**	**Number of Valid**
58	0	8	50
Bracketing Recall	**Bracketing Precision**	**Bracketing FMeasure**	**Complete match**
88.71	88.71	88.71	68.00
No Crossing	**2 or less Crossing**	**Tagging Accuracy**	**Average Crossing**
80	92	99.05	0.48

Improvement

Number of Sentence	**Number of Error**	**Number of Skip**	**Number of Valid**
58	0	2	56
Bracketing Recall	**Bracketing Precision**	**Bracketing FMeasure**	**Complete match**
87.46	87.46	87.46	62.50
No Crossing	**2 or less Crossing**	**Tagging Accuracy**	**Average Crossing**
78.57	92.86	97.19	0.5

Evaluation:

As the number of parsed sentences grew, recall and precision slightly decreased, but still remained in the 87-zone.

Problems and issues:

1. When choosing what solution to base hw3 on, we favoured the one that could easily retrieve the parses. Although we would have to deal with nltk format later, we thought it would still save us time.

2. We had a discussion about whether the grammar is in cnf and at what stage we need to check this if at all.

3. Filling the matrix correctly was more difficult because of an elaborate data structure that we had now. Every cell contained a tuple of (tree, probability), getting more complex with each iteration. We spent some time figuring out how to loop through it correctly and how to address the elements.

4. Adjusting the output of the final best parse to an appropriate format for *evalb* was tricky. The solution was the recursive method *parseTree* which would descend the levels of the tree and retrieve the nodes (and terminals, when it came across a pair of quotation marks).

5. Comparing our first *evalb* results to those posted on GoPost, we realized that one sentence did not parse, because the terminal never got into the dictionary. Unlike all the other terminals *“ 'd ”* was surrounded by double, not single quotes, so it was skipped because of how we were stripping the quotes and filling the dictionary. Hard to find, easy to fix.

6. To parse the sentences containing unknown words, we had to create a list of non-terminals that can possibly generate terminals, and used a special valueless dictionary for non-terminals to speed up the search. For words not in dictionary, we automatically created trees with all possible tags; unneeded trees would be pruned on one of the next levels, but we would expect to build at least one correct parse.

7. Because of how we assigned our probabilities to the additional trees, they don't add up to 1 anymore. We discussed ways to offset that (mainly thinking of add-1 smoothing and interpolation), but couldn't come up with a way to implement it here.

8. We did not manage to fix parses for two sentences: *I would like to travel to Westchester* . and *What flights go from ?* We considered tweaking the grammar or forcing a likely parse, but we could see that all the ideas we could come up with would inevitably affect the parses we already have and would result in accepting ungrammatical sentences.