Elements of Language Processing and Learning
Assignments 1 and 2

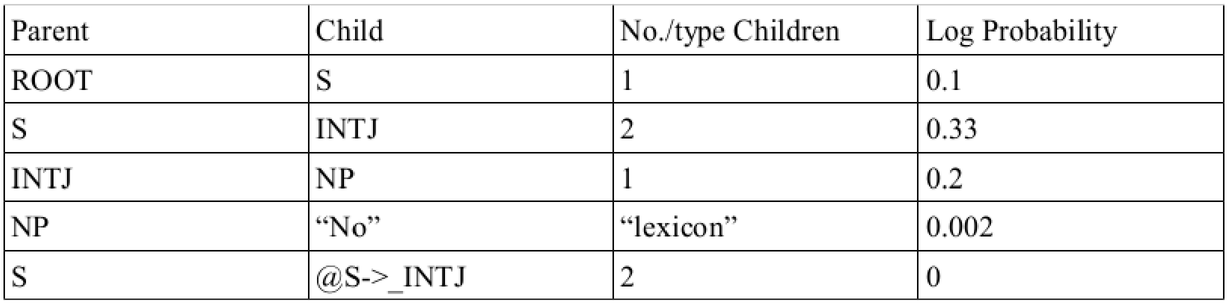
Liveris Avgerinidis 10318828
Jaysinh Sagar 10318771
Ioannis Zervos 10333630

Assignment 1: Computing the probability of a Tree

For this assignment, we are asked to compute the probability of a given sentence tree. In order to do this, we use rule look up table for parent->child node pairs and then retrieve the learned probability of that pair. For this, we consider 3 types of node relations. First, being where each parent has only 1 child where we apply unary parent-child rule. Another type of relation we have is when a parent has 2 children for which we apply the binary parent-children rules. Lastly, we have a third type of relation where the parent has a child which is a leaf node of that tree illustrating a natural language word of the sentence where we apply lexicon rules. Based on the presence of each of these parent-child relations in a given tree, we can easily compute the trees overall probability.

The first step for us was to find a way to traverse through the annotated tree structure in order to get word-token pairs or parent-child pairs. For this, we try an iterative approach as well as a recursive approach. Both of these approaches work well in our case where in our submitted implementation we have the recursive method. For traversal over the tree, we implement a method called traversalAnnotatedTree in the baselineCkyParser.java class where we take in an annotated tree and generate a matrix of parent->child pairs along with the type of child with that relations log probability. We illustrate a part of this table below in Table 1. Once we generate this matrix for the tree, we simply add up over all log probabilities we attain from 2 custom methods we define in the grammar.java class, getUnScore and getBiScore. Each of these functions matches pairs of given parent and child labels and returns the logged score of that pair.

Parent Child No./type Children Log Probability



Parent	Child	No./type Children	Log Probability
ROOT	S	1	0.1
S	INTJ	2	0.33
INTJ	NP	1	0.2
NP	“No”	“lexicon”	0.002
S	@S->_INTJ	2	0

Table 1. Matrix table of tree token information.

From our experiment, we draw results for test sentences shown in table 2. We also notice some -infinity values which represent absence of the probability of a certain rule for a parent-child pair.

```
0: logscore =-50.99367882096118
1: logscore =-Infinity
2: logscore =-39.85517820793024
3: logscore =-41.67112250048548
4: logscore =-92.5165915926479
5: logscore =-Infinity
6: logscore =-101.62447403037062
7: logscore =-68.99276774663788
8: logscore =-55.708379070109174
9: logscore =-57.39394243941492
Total log prob:-Infinity
```

Assignment 2.1: Adding support of unary rules in CKY algorithm

CKY algorithm is a bottom-up parsing algorithm for context-free grammars given in chomsky normal form (CNF)[1]. This algorithm makes use of an assisting structure , called chart, where we keep information of the rules that we have used and the respective score. This chart is an upper triangular table, whose size equals the length of the tokens in the sentence.

For this assignment we were provided with an implementation of CKY algorithm, which doesn't include the unary rule checks. In order to add support for the unary rules, the following methods are extended/added:
-the class 'EdgeInfo', new field unary rule is added to keep unary rules
-the method 'set' of class chart is changed to add new best score
-the method 'setBackPointer' changed to support unary rules
-the methods 'getBestParse' and 'traverseBackPointersHelper' are extended to support unary rules

a) Creating the chart
The first step is to fill in the diagonal of this chart, using the preterminal rules of the grammar and consequently to check if there is any unary rule with child the parent of these preterminal rules, which we also add in the diagonal. The positions of the chart that are parallel to the diagonal and represent all the 2-grams in our sentence, are firstly filled using the binary rules. This is done by checking if there is any binary rule with left child any of the parents of the rules of its left position in the chart and right child any of the parents of the rules of the position just below it (left-to-right, bottom-up). After this is done, one more check for unary rules is performed. Unary rules which have children any of the parents of these binary rules are kept. The unary rules in our implementation were found using the unaryClosure which is a precomputed lookup table that

returns all possible unary chains. The next steps are similar with binary and unary checks described above, but this time all the possible combinations (according to the span) are examined to form 3-word constituents, 4-word constituents, etc. and additional information of the best splitting/mid-point is kept using backpointers.

b) Creating the backpointers

The backpointer concept in the CYK algorithm generates all possible parent-node combinations for each word token tracing back to the root node “guessing” the probably tree structure. This is done by finding the best matching parent-child combinations for terminal nodes (each word) to rules for subsequent nodes. This eventually creates a probable tree by selecting the best score between each subsequent higher j cell between the diagonal i cells of the chart. This is best illustrated in the link [[1]] [[2]]as well as the lecture notes for this topic in class.

c) Generating the tree

In order to create the tree, we make use of the traverse back pointer implementation is two parts; the traverse backpointer method provided in the nlp framework in the assignment and the traverse back pointer helper method that we modify to take into consideration unary rules as well as the existing binary rules configuration. The helper function we define is illustrated in three primary steps:-

1. We check for Unary rules present. If so, get a list of the unary path from the UnaryClosure class and iterate through it setting unary rule based parent-child trees

2. Else if no such unary rules exist, we check for presence of binary rules and traverse through the input tree to the left and right child and adding children to the parent nodes recursively. This function was given to us in the framework.

3. Else If child is terminal, apply terminal rule and add terminal child.

The traverse back pointers method provided calls the helper function and traverses through the back pointers to create the tree.

Results:

With our changes we have the following results for the 1034 sentences of the test set:

Precision	Recall	F1	Ex
79.63	70.51	74.79	15.08

[1] http://www.youtube.com/watch?v=MiEKnFyErbQ

[2] http://www.diotavelli.net/people/void/demos/cky.html

Conclusion:

We couldn’t reproduce the results reported in the assignment instructions. We think that there is one bug in our implementation when we assign the unary rules. Even after several evaluations and tracing, the implementation seems correct but often, the rules selected for the chart cell are incorrect (based on the score) leading to incorrect tree generation. We are still yet to find where there is a fallacy or a bug in our code or if something is implemented incorrectly if that is the case, but are yet to find something that would give the EX score a boost to ~23 while the F1 in all our experiments was a decent 76-77 range.

References

[1]http://en.wikipedia.org/wiki/CYK_algorithm