University of Amsterdam

Elements of Language Processing and Learning

Project 1: Statistical Parsing, Steps 1 & 2 25 November, 2012

By:
Georgios Methenitis, 10407537
Marios Tzakris, 10407537

1 Introduction

In this project we had to implement a prototype statistical parser. Having the given treebanks we had to parse and extract the rules using this statistical parser (Step 1), resulting to a probabilistic context free grammar (PCFg). Then, given this extracted grammar, we implement the Cocke-Younger-Kasami (CYK) algorithm, in order to parse some test sentences generating a parse-forest for each one of them. In Section 2, we are going to describe the implementation of our parser. In Section 3, we are going to describe the implementation of the CYK algorithm as well as, our assumptions in respect to the unknown words, and a brief explanation of every step of this algorithm through pseudo-code. Section 4, serves as an epilogue to the first two steps of this project.

2 Statistical Parsing

Statistical parsing is a parsing method for natural language processing. The key idea behind this procedure, is the association between grammar rules from the linguistics point of view with probabilities of each one of them. Make it feasible for us, to be able to parse, and compute the probability of a complete parse of a sentence.

2.1 Stochastic/Probabilistic context-free grammar (SCFG or PCFG)

A probabilistic context-free grammar is a set of rules in which each production is augmented with a probability. Each rule is represented by the non-terminal left-hand side node, and the right-side node or nodes. In our particular data-set there are only binary and unary rules. A typical example of binary rule is this: $X \rightarrow \alpha\ \beta$. In general in our treebanks there were mostly binary rules but in some cases we had to handle and unaries which have this form: $X \rightarrow \alpha$. Each rule has a probability and this probability is given by: $P(X \rightarrow \alpha|X)$, or $P(X \rightarrow \alpha\beta|X)$, for the above two examples of rules. So, we can derive that the probabilty of each rule is given by the frequency of the right hand nodes, given the left hand nodes of a rule. Assuming that, the frequency of each specific rule is denoted by f_r, the non-terminal nodes by, N_i, and finally the productions which are a sequence of one or more non-terminal or terminal nodes by, n_i, we have:

$$P(N_i \rightarrow n_i|N_i) = \frac{f_r(n_i|N_i)}{\parallel N_i \parallel}$$

, where $f_r(n_i|N_i)$, is the number of times that n_i is produced by the non-terminal node N_i during the training.

Algorithm 1 Rule Production

```
1: Input: $Nodes\_Table$ (1)
2: Output: $Rules$
3:
4: for i ∈ $(0, \textbf{length}(Nodes\_Table) - 1)$ do
5:     $rightSide = \emptyset$
6:     for j ∈ $(i + 1, \textbf{length}(Nodes\_Table))$ do
7:         if $(Nodes\_Table[j].level - 1 == Nodes\_Table[i].level)$ then
8:             if $(rightSide == \emptyset)$ then
9:                 $rightSide \to Nodes\_Table[j].rule$
10:            else
11:                $rightSide \to rule, Nodes\_Table[j].rule$
12:            end if
13:        end if
14:        if $(j == \textbf{lenth}(Nodes\_Table) - 1$ or $Nodes\_Table[j].level == Nodes\_Table[i].level)$ then
15:            if $(rightSide == \emptyset)$ then
16:                save_rule$(Rule(Nodes\_Table[i].rule, rightSide))$
17:            end if
18:        end if
19:    end for
20: end for
```

2.2 Implementation of the PCFG parser

To parse the given treebank and extract the rules with their probabilities, we did not use a third-party library, instead we built our own parser. Given the treebank's form,

```
(TOP (S (NP (NNP Ms.) (NNP Haag)) (S@ (VP (VBZ plays) (NP (NNP Elianti))) (. .))) )
```

we parse tree of the training set, keeping nodes and a level information according to the parenthesis level about each one of them. After this procedure our parsing table (1) looks like this:

TOP	0	VP	3
S	1	VBZ	4
NP	2	plays	5
NNP	3	NP	4
Ms.	4	NNP	5
NNP	3	Elianti	6
Haag	4	.	3
S@	2	.	4

Algorithm 1, presents in pseudo-code this procedure. This algorithm starts from each node heading towards the end of this list until it comes along with a node of the same level or with the end. In this process if there is a node with level one more than the current node, this node automatically infers that it is its child. The result rules derived by the above example sentence are following:

TOP → S	VP → VBZ NP
S → NP S@	VBZ → plays
NP → NNP NNP	NP → NNP
NNP → Ms.	NNP → Elianti
NNP → Haag	S@ → VP .
. → .	

Algorithm 2 CYK Algorithm

```
 1: Input: $RulesRL[]$, $Sentence\ S[]$
 2: $\#RulesRL$, is a dictionary in which for every production $p$, gives $RulesRL[p]$ non-terminal nodes which have
    this production.
 3: Output: $Chart[\mathbf{length}(S)+1][\mathbf{length}(S)+1]$
 4: for i ∈ (0,length(s)) do
 5:    if $S[i] \notin RulesRL[]$ then
 6:       handle_unknown_word($S[i]$)
 7:    else
 8:       for node ∈ $RulesRL[S[i]]$ do
 9:          $Chart[i][i+1]$.add($node$)
10:       end for
11:    end if
12:    check_unaries($Chart[i][i+1]$)
13: end for
14: $n = \mathbf{length}(S) + 1$
15: for span ∈ (2,n) do
16:    for begin ∈ (0,n-span) do
17:       $end = begin + span$
18:       for split ∈ (begin,end) do
19:          for nodeL ∈ $chart[begin][split]$ do
20:             for nodeR ∈ $chart[split][end]$ do
21:                for node ∈ $RulesRL[nodeL, nodeR]$ and node $\notin Chart[begin][end]$ do
22:                   $Chart[begin][end]$.add($node$)
23:                end for
24:             end for
25:          end for
26:       end for
27:       check_unaries($Chart[begin][end]$)
28:    end for
29: end for
```

2.3 Results

After running our parser in the complete training set, we got all rules, the frequency for every production of each rule, and the number of appearances for every non-terminal node. From this data, we extracted the complete PCFG grammar from the training set in a file with the following form:

```
<id>   <Non-Terminal>  <Production>       <Probability>
00000       TOP         FRAG%%%%NP      0.000225954658432
00457       NP          ADVP NNS        6.53558462438e-06
```

3 CYK Algorithm

In computer science, the Cocke-Younger-Kasami (CYK) algorithm, is a parsing algorithm for context-free grammars. It employs bottom-up parsing and dynamic programming, considering all possible productions that can generate a sentence. In this algorithm, we want to fill a chart with non-terminal nodes in order to be able to consider all possible trees. The CKY algorithm scans the words of the sentence attempting to add non-terminal nodes to its chart. For each production of a non-terminal node that is matched from the PCFG grammar we add this left side node to the chart. To improve its performance we didn't iterate on all possible non-terminal nodes, instead we have used a dictionary to index every node added to the chart.

Algorithm 3 Unaries Handling

```
1: Input: $Chart[i][j]$
2: Output: Updated $Chart[i][j]$
3: $added = true$
4: while $added$ do
5:     $added = false$
6:     for ref_node $\in chart[i][j]$ do
7:         for node $\in RulesRL[ref\_node]$ and node $\notin Chart[i][j]$ and node== $"TOP"$ do
8:             $Chart[i][j]$.add($node$)
9:             $added = true$
10:        end for
11:    end for
12: end while
```

3.1 Implementation of the CYK Algorithm

Algorithm 2, presents our algorithm in pseudo-code. As you can realize, there are some differences to the algorithm as it is described in the *Stanford* slides. This happens because we make use of the dictionaries in python to store the nodes in every chart position. So, we don't iterate over all non-terminal nodes but only over the existing in each chart cell.

4 Conclusion

In this first two steps, we have seen....