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

Input: $Nodes\_Table$ (1)
Output: $Rules$

for i $\in$ $(0, \textbf{length}(Nodes\_Table) - 1)$ **do**
  $rightSide = \emptyset$
  for j $\in$ $(i + 1, \textbf{length}(Nodes\_Table))$ **do**
    if $(Nodes\_Table[j].level - 1 == Nodes\_Table[i].level)$ **then**
      if $(rightSide == \emptyset)$ **then**
        $rightSide \rightarrow Nodes\_Table[j].rule$
      else
        $rightSide \rightarrow rule, Nodes\_Table[j].rule$
      end if
    end if
    if $(j ==$ **lenth**$(Nodes\_Table) - 1$ **or** $Nodes\_Table[j].level == Nodes\_Table[i].level)$ **then**
      if $(rightSide == \emptyset)$ **then**
        save_rule$(Rule(Nodes\_Table[i].rule, rightSide))$
      end if
    end if
  end for
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 .
. → .	

Table 1: Syntactically Ambiguous Words

Word	Non-Terminal	Probability
down	RP	0.080391
	RB	0.011398
	NN	0.000008
	RBR	0.000566
	VBP	0.000080
	JJ	0.000163
	IN	0.001441
that	NN	0.000008
	VBP	0.000080
	WDT	0.462273
	RB	0.000710
	IN	0.048856
	DT	0.014272
's	VBZ	0.056386
	PRP	0.000459
	NNS	0.000017
	POS	0.928514
	NNP	0.000011
a	DT	0.235392
	FW	0.029915
	LS	0.027778
	SYM	0.172414
	NNP	0.000022

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
```

2.3.1 Syntactically Ambiguous Words

To find find the syntactically ambiguous words we used every terminal word and count by how many non-terminal is been produced. Table 1, presents some of the most ambiguous syntactically words in out training data.

2.3.2 Most Likely Productions

To find the most likely productions for the set of non-terminals {V P, S, NP, SBAR, PP}, we store the dictionary in respect to the probability of each of those productions and print the four with the biggest probability. Table 2.3.2, presents these results.

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.

Table 2: Most Likely Productions

Non-Terminal	Production	Probability
NP	DT, NP@	0.1284
	NP, PP	0.1111
	DT, NN	0.0958
	NP, NP@	0.0888
PP	IN, NP	0.7963
	TO, NP	0.0787
	IN, S%%%%%VP	0.0263
	IN, NP%%%%%QP	0.0128
SBAR	IN, S	0.4537
	WHNP, S%%%%%VP	0.2671
	WHADVP, S	0.0904
	WHNP, S	0.0665
VP	VBD, VP@	0.0767
	VB, NP	0.0663
	VB, VP@	0.0640
	MD, VP	0.0591
S	NP, S@	0.3541
	NP, VP	0.3439
	PP, S@	0.0715
	S, S@	0.0615

Algorithm 2 CYK Algorithm

Input: $RulesRL[]$, $Sentence\ S[]$
$\#RulesRL$, is a dictionary in which for every production p, gives $RulesRL[p]$ non-terminal nodes which have this production.
Output: $Chart[\textbf{length}(S)+1][\textbf{length}(S)+1]$
for i $\in$ (0,**length**(s)) **do**
  if $S[i] \notin RulesRL[]$ **then**
    handle_unknown_word$(S[i])$
  else
    for node $\in RulesRL[S[i]]$ **do**
      $Chart[i][i+1]$.**add**$(node)$
    end for
  end if
  check_unaries$(Chart[i][i+1])$
end for
$n = \textbf{length}(S)+1$
for span $\in$ (2,n) **do**
  for begin $\in$ (0,n-span) **do**
    $end = begin + span$
    for split $\in$ (begin,end) **do**
      for nodeL $\in chart[begin][split]$ **do**
        for nodeR $\in chart[split][end]$ **do**
          for node $\in RulesRL[nodeL, nodeR]$ **and node** $\notin Chart[begin][end]$ **do**
            $Chart[begin][end]$.**add**$(node)$
          end for
        end for
      end for
    end for
    check_unaries$(Chart[begin][end])$
  end for
end for

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 ∈ chart[i][j] do
7:       for node ∈ RulesRL[ref_node] and node ∉ 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. In this algorithm the first 13 lines of code is for initialization of the chart. In this part, we are looking for unary rules with productions each word of the sentence. If a word does not exist in our rules then we handle it differently and we are going to discuss it later in this report. After this step, we check every updated chart position for unaries that may exist. Algorithm 3, presents the unary handling we have done. According to the algorithm in the *Stanford* slides, we check all nodes in a chart position as productions from another rules and we add these non-terminals to the same chart position. This is done until there is no unary rule to be added.

3.2 Unknown Words

As we have seen in the test sentences, there are plenty of words which were not able to associate themselves to a rule form the PCFG grammar. We have tried to with a simple way to associate them in a rule. The following Table 3.2, illustrates this procedure with a few examples. In this way, there

Format	**Examples**	**Type**	**Non-Terminal**
number, floats	12.3, 12-3-2012	numerical	CD
*ly, *y	fully, difficulty	adverb	ADV
*ing, *ion, *ist(s), *(e)s, *er(s)	criterion, linguistic	noun	N
-, *able, *ed	capable, drug-seeking	adjective	ADJP
else	ALFO, L.F	$noun^{p=0.2}$, $name^{p=0.8}$	N, NNP

are not unknown words anymore and every word can be assigned as a production of a non-terminal node. Is it obvious,that it is a very simplified way of recognize the type of a word, but we are sure that we are able to build a better system to recognize the type of unknown words, with a more stochastic way, and in respect to the training set.

3.3 Results

Our implementation of the CYK algorithm is well optimized and it is neither time or space consuming speaking about its complexity. For the time being, we have not deal with probabilities during the process of building the parse-forest. So, it is normal, that it the algorithm is not very fast due to the fact that it is constructing all possible trees that may have generate each sentence.

3.3.1 TOP Productions

The covering productions can be generated from an input sentence given the grammar. For example, assuming that we have the sentence:

"*No, it wasn't Black Monday.s*"

The productions `TOP` $\rightarrow$ `X P` that cover the whole sentence in the entry $chart[0, n]$, after running our CYK parser on this sentence was:

`TOP` $\rightarrow$ `FRAG`
`TOP` $\rightarrow$ `FRAG%%%%%ADVP`
`TOP` $\rightarrow$ `PP`
`TOP` $\rightarrow$ `NP%%%%%S`
`TOP` $\rightarrow$ `SBAR`
`TOP` $\rightarrow$ `UCP`
`TOP` $\rightarrow$ `SINV`
`TOP` $\rightarrow$ `ADJP`
`TOP` $\rightarrow$ `SQ`
`TOP` $\rightarrow$ `PRN`
`TOP` $\rightarrow$ `SBARQ`
`TOP` $\rightarrow$ `FRAG%%%%%SBAR`
`TOP` $\rightarrow$ `ADVP`
`TOP` $\rightarrow$ `VP`
`TOP` $\rightarrow$ `S`
`TOP` $\rightarrow$ `NP`
`TOP` $\rightarrow$ `X`
`TOP` $\rightarrow$ `FRAG%%%%%NP`
`TOP` $\rightarrow$ `S%%%%%VP`
`TOP` $\rightarrow$ `X%%%%%SBARQ`

4 Conclusion

In this report we have presented an simple but efficient way to extract the PCFG from a treebank data file. Our implementation makes use of the *Python* dictionaries in order to store the rules of the grammar in efficient way in order to be used in the most optimized by the CYK algorithm which generates a parse forest for a sentence based on an existing grammar. Additionally, we have discussed how we implemented the CYK algorithm and how we handled the unknown words which are existed in the test sentences data. More information about the code itself, you can find in the readme file attached with our source code.