University of Amsterdam

Elements of Language Processing and Learning

Project 1: Statistical Parsing 25 November, 2012

By:
Georgios Methenitis, 10407537
Marios Tzakris, 10406875

1 Introduction

In this project we implemented a prototype statistical parser. Having the given treebank data file we parsed and extracted the rules using this statistical parser (Step 1), resulting to a probabilistic context free grammar (PCFg). Then, given this extracted grammar, we implemented 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 discuss about our implementation of the CYK algorithm as well as, our assumptions in respect to the unknown words, the unary rules, and a brief explanation of every step of this algorithm through pseudo-code and examples. Section 6,

Finally, section 5, 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 for each one of them. Make it feasible for us, to be able to analyze, 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 the 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 likelihood 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. In addition, $\parallel N_i \parallel$, is the number of production this non-terminal have.

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 parsing library, instead we built our own parsing algorithm. Given the treebank's form,

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

we parse each tree of the training set, keeping nodes and a piece of informationabout the level of each node according to the parenthesis level. After this procedure our $Nodes\_Table$ (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

After the generation of the $Nodes\_Table$, it is time to generate the rules derived from this tree. 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 resulted 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 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 our training data.

2.3.2 Most Likely Productions

In addition, to compute the most likely productions for the set of non-terminals {V P, S, NP, SBAR, PP}, we sort the dictionary in respect to the probability of each of those productions and print the four with the biggest probability. Table 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 produce all possible trees. The CYK 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

3.1 Implementation of the CYK Algorithm

Algorithm 2, presents our implementation of the CYK 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 15 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. We are going to discuss this later in this report. Then and 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 their-selves to a rule from the PCFG grammar. We have tried to deal with it with a simple approach, associating each one of them with a rule. The following Table 3.2, illustrates this procedure with a few examples. In this way, there are not unknown words anymore and every word can be assigned as a

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

production of a non-terminal node. Is it obvious,that it is a very simplified way of recognize the type

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[length(S) + 1][length(S) + 1]

# Chart initialization
for i ∈ (0,length(s)) do
    if S[i] ∉ RulesRL[] then
        handle_unknown_word(S[i])
    else
        for node ∈ RulesRL[S[i]] do
            Chart[i][i + 1].add(node)
        end for
    end if
    check_unaries(Chart[i][i + 1])
end for

# Fill productions in chart
n = length(S) + 1
for span ∈ (2,n) do
    for begin ∈ (0,n-span) do
        end = begin + span
        for split ∈ (begin,end) do
            for nodeL ∈ chart[begin][split] do
                for R ∈ chart[split][end] do
                    for node ∈ RulesRL[nodeL, R] and node ∉ 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
```

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*"

Algorithm 3 Unaries Handling

Input: $Chart[i][j]$
Output: Updated $Chart[i][j]$
$added = true$
while $added$ **do**
  $added = false$
  for ref_node $\in chart[i][j]$ **do**
    for node $\in RulesRL[ref\_node]$ **and node** $\notin Chart[i][j]$ **and node is** "TOP" **do**
      $Chart[i][j]$.**add**($node$)
      $added = true$
    end for
  end for
end while

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

4 Viterbi

In this section, we are going to present our Viterbi implementation which we built in order to produce the most probable tree for each input sentence. We have already, a Probabilistic context-free grammar extracted for the training sentences (Step 1), and a CYK implementation which gives us a parse-forest for each sentence in the test sentences data (Step 2). From this parse-forest we can build all possibles trees for each sentence.

4.1 Most-probable Parse Tree

In order to implement an efficient Viterbi algorithm, we had to reconstruct the CYK algorithm in order to fill its chart with the most probable parse-tree. To do that, we keep in each position of the chart the most-probable production for each left hand side node that appears in the each position. So, each position of the chart is now holding different left hand nodes, each one of them leads us to the most probable production, not to all possible productions. Algorithm 4, presents only the changes in the pseudo-code for the alternative implementation of the CYK-algorithm. Additionally, in the same manner, we changed the way that handle unary productions in each chart position. This is illustrated

Algorithm 4 CYK Algorithm for Most-probable Production

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: $Chart[\textbf{length}(S)+1][\textbf{length}(S)+1]$

Chart initialization
...
for node $\in RulesRL[S[i]]$ **do**
  if $node \notin Chart[i][i+1]$ **then**
    $Chart[i][i+1][node] = (S[i], p(node \rightarrow S[i]))$
  else
    if $p(node \rightarrow S[i]) > Chart[i][i+1][node].prob$ **then**
      $Chart[i][i+1][node] = (S[i], p(node \rightarrow S[i]))$
    end if
  end if
end for
...

Fill productions in chart
...
for node $\in RulesRL[L, R]$ **do**
  if $node \notin Chart[b][e]$ **then**
    $Chart[b][e][node] = (L, R, p(node \rightarrow L, R) \times Chart[b][s][L].prob \times Chart[s][e][R].prob, s)$
  else
    if $p(node \rightarrow L, R) \times Chart[b][s][L].prob \times Chart[s][e][R].prob > Chart[i][i+1][node].prob$ **then**
      $Chart[b][e][node] = (L, R, p(node \rightarrow L, R) \times Chart[b][s][L].prob \times Chart[s][e][R].prob, s)$
    end if
  end if
end for
...

Algorithm 5 Unaries Handling for Most-probable Production

...
for node $\in RulesRL[ref\_node]$ **do**
  if $node \notin Chart[i][j]$ **then**
    $Chart[i][j][node] = (node, p(node \rightarrow ref\_node) \times Chart[i][j][ref\_node].prob)$
    $added = true$
  else
    if $p(node \rightarrow ref\_node) \times Chart[i][j][ref\_node].prob > Chart[i][j][node].prob$ **then**
      $Chart[i][j][node] = (node, p(node \rightarrow ref\_node) \times Chart[i][j][ref\_node].prob)$
      $added = true$
    end if
  end if
end for
...

in the Algorithm 5. From all productions that can be produced by a left hand side node, we keep only the most probable. This different approach will help us save computational cost during Viterbi algorithm. Imagine, that, if we had kept the old approach, we should have iterate over all possible productions in each non-terminal node to find the most probable one, every time.

4.2 Viterbi Algorithm Implementation

Viterbi algorithm builds the most probable tree which produced a specific sentence in a recursive way. We always start from the upper right corner of the chart with the `TOP` node. Next, we follow the child node(s) of the `TOP` node and call the same algorithm for the child node(s). Algorithm 6, presents the pseudo-code of the algorithm that we design and build for this purpose. The output of this algorithm is list of pre-order visited nodes a label which defines the level of each node. Additionally, we are taking care of the nodes labeled with `"%%%%"`, or `"@"`. For the first case, `"%%%%"`, we store the first node in the unary production with the predefined level (one more than its parent), and the second node with the same level plus one. In addition, child(ren) of this node will be called with one more level than the original. For the second case, `"@"`, we just do not store the node and we call its child(ren) with

Algorithm 6 Viterbi

Input: $chart[][], x, y, node, words, level$
Initial values: $chart[][], 0, length(words) - 1, TOP, words, -1$
Output: $nodes[]$
if "@" $\in node$ **then**
  $level- = 1$
else
  if "%%%%" $\in node$ **then**
    $temp = node.split(unary_node)$
    $nodes.append(temp[0], level, False)$
    $nodes.append(temp[1], level + 1, False)$
    $level+ = 1$
  else
    $nodes.append(tree_node(node, level, False))$
  end if
end if
if $node \in chart[x][y]$ **then**
  $n = chart[x, y][node]$
  if $(x + 1) == y \cup n.child == node \cap n.child \in words$ **then**
    $nodes.append(n, level + 1, True)$
  end if
  if n *is unary* **then**
    $Viterbi(chart, x, y, n, words, lvl + 1)$
    return
  else
    $Viterbi(chart, x, n.split, n[l], words, level + 1)$
    $Viterbi(chart, n.split, y, n[r], words, level + 1)$
    return
  end if
else
  return
end if

the original level subtracted by one. In this way, child(ren) nodes will end up being child(ren) of their grandparent, not of their father. We can visualize this list of nodes for the example sentence:

"The collateral is being sold by a thrift institution . "

as the following: ,where each node has a label of its level. In this way, we can build the tree of the most

TOP 0	VP 2	VBN 5	a 8
S 1	VBZ 3	sold 6	NN 7
NP 2	is 4	PP 5	thrift 8
DT 3	VP 3	IN 6	NN 7
The 4	VBG 4	by 7	institution 8
NN 3	being 5	NP 6	. 2
collateral 4	VP 4	DT 7	. 3

probable derivation of the tree easily. Algorithm 7, presents this procedure in pseudo-code. Having, linear complexity builds the tree in the same as Penn WSJ format. Although, that printing directly the tree from viterbi algorithm was feasible we preferred doing this simple algorithm in order for the two functions to be separate. The most probable tree production for the above sentence is given in Penn WSJ format:

Algorithm 7 Build Tree

```
1: Input: nodes
2: Output: tree
3: level = −1
4: while i < length(nodes) do
5:     if nodes[i].level ≥ level then
6:         if nodes[i].terminal : then
7:             tree + = nodes[i].node + ")"
8:         else
9:             tree + = "(" + nodes[i].node
10:            i+ = 1
11:        end if
12:        level = nodes[i].level
13:    else
14:        tree + = ")"
15:        level − = 1
16:    end if
17: end while
18: for i = 1 to level do
19:    tree + = ")"
20: end for
```

```
(TOP(S(NP(DT The)(NN collateral))(VP(VBZ is)(VP(VBG being)(VP(VBN sold)(PP(IN by)(NP
(DT a)(NN thrift)(NN institution))))))(. .)))
```

4.2.1 Unknown Words

As we seen before, in Section 2, we came up with a simple approach of handling unknown words. We used the same approach here. A better approach would have been to train a function for assigning unknown words to left hand side nodes while training. This would have been a better and more probabilistic approach than the one we have now, and hopefully, would have given better results.

4.2.2 Long Sentences

Test sentences include sentences of different lengths (number of words). We have been asked to evaluate only those with less or equal length to 15. So, we filtered out the long sentences, instead we are writing an empty tree like the following:

```
(TOP (POS word_1) (POS word_2)....(POS word_n))
```

4.3 Results

For evaluating our trees against the gold test trees, the correct ones, we used EvalC software which is written in *Java*, by Federico Sangati. We evaluate our trees against the ones from the gold tree set.

5 Conclusion

In this report we have presented a 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 efficiently in order to be used in the most optimized way 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.