Elements of Language Processing and Learning Project 1: Statistical Parsing and CYK*

Peter Lubell-Doughtie, 6095445 and Davide Modolo, 6209033

University of Amsterdam

1 Introduction

In this report we describe our implementation of a procedure to extract probabilistic context free grammars (PCFGs) from a Treebank and then, given a PCFG, parse sentences using the Cocke-Younger-Kasami (CYK) algorithm. We begin by providing an introduction to PCFGs and the CYK algorithm in Sec. 2, then discuss implementation details in Sec. 3. In Sec. 4 we discuss our results and finally we conclude in Sec. 5.

2 Statistical Parsing

The field of statistical parsing is concerned with generating grammars and parsing sentences by apply automated methods to real world text corpuses. An existing bracketed corpus is used as a training set to build the grammar, often expressed as a context free grammar that may or may not be in Chomsky normal form. The important advantage which statistical grammar creation has over hand crafted grammars is that in statistical grammar creation a grammar is created *for* the data such that every encountered bracketed sentence can be generated by the grammar. Whereas, with hand crafted grammars, the data need not fit the created grammar. This implies that generated grammars are necessarily capable of making better use of the existing data because each production in each bracketed sentence will be accounted for in a generated grammar.

2.1 Probabilistic Context Free Grammars (PCFGs)

Put simply, a probabilistic context free grammar (PCFG) is a context free grammar in which probabilities are assigned to each rule. These probabilities represent the likelihood of a particular rewrite amongst the set of possible rewrites and the sum over all possible rewrites must equal unity. Formally, a PCFG G consists of:

- a set of terminals, $\{w^k\}, k = 1, \cdots, V$
- a set of nonterminals, $\{N^i\}, i = 1, \cdots, n$

* This report completes steps 1-2 of project 1 for *Elements of Language Processing and Learning* 2010, professor Khalil Sima'an, teaching assistant Gideon Wenniger.

- a start symbol, N^1, which we define as TOP
- a set of rules, $\{N^i \rightarrow \zeta^j\}$, where ζ^j is a sequence of terminals and nonterminals.
- and a set of probabilities $P(N^i \rightarrow \zeta^j | N^i)$ for each rule such that:

$$\forall i \left[\sum_j P(N^i \rightarrow \zeta^j | N^i) = 1 \right] \tag{1}$$

The probabilities $P(N^i \rightarrow \zeta^j | N^i)$ are defined as the relative frequency of the production $N^i \rightarrow \zeta^j$. Given a treebank $\overline{\mathbf{tb}}$, which contains the set of nonterminals and terminals, we can create the set of productions, $\mathcal{R}$, and a frequency function, $\mathcal{F}_\mathcal{R} : \mathcal{R} \rightarrow \mathbb{N}^+$, which maps from productions to non-zero natural numbers representing the frequency of each rule.

We collect these productions and the frequency function into a multi-set $\mathbf{\Pi}_{\overline{\mathbf{tb}}} = \langle \mathcal{R}, \mathcal{F}_\mathcal{R} \rangle$. The relative frequency, $rf(\cdot, \cdot)$, of a specific production and $\mathbf{\Pi}_{\overline{\mathbf{tb}}}$ is then defined as:

$$rf(N^i \rightarrow \zeta^j, \mathbf{\Pi}_{\overline{\mathbf{tb}}}) = \frac{\mathcal{F}_\mathcal{R}(N^i \rightarrow \zeta^j)}{\sum_{N^i \rightarrow \zeta^k \in \mathcal{R}} \mathcal{F}_\mathcal{R}(N^i \rightarrow \zeta^k)} \tag{2}$$

where the frequency of one production $N^i \rightarrow \zeta^j$ is represented in the numerator and the denominator normalizes this frequency by summing over all productions from N^i to any sequence of terminals and nonterminals. Based upon this we define the probability of a production as its relative frequency:

$$P(N^i \rightarrow \zeta^j | N^i) \equiv rf(N^i \rightarrow \zeta^j, \mathbf{\Pi}_{\overline{\mathbf{tb}}}) \tag{3}$$

for each nonterminal N^i and terminals and nonterminals ζ^j. By construction we have ensured that, for all nonterminals N^i in our data set, $P(N^i \rightarrow \zeta^j | N^i) \in (0, 1]$ and that Eq. 1 holds.

Given a PCFG we will be able to generate all possible parses for a sentence by determining which productions can generate each terminal word in the sentence, then which productions can generate these found nonterminals, and so on. We can further determine the likelihood of a particular parse by examining the probabilities for the application of each production.

2.2 Parsing with the CYK Algorithm

The CYK algorithm operates by considering all possible productions that could have generated the sentence and then, for each adjacent pair of nonterminals A and B, considering all possible nonterminals N for which there is production $N \rightarrow A\ B$. Assuming we are given a sentence $S = w_0 \dots w_{n-1}$, our goal is to fill a chart matrix $\mathcal{C}$ with nonterminals such that if there is a nonterminal $N \in \mathcal{C}(i, j)$ this nonterminal spans the words $i \dots j$, where each entry in the matrix is a set.

We can divide this process into an initialization step and a deduction, or generation, step. In the initialization step the CYK parser considers all $0 \leq i <$

n and adds a nonterminal A and associated terminal w_i to $\mathcal{C}(i, i+1)$ for all productions such that $A \rightarrow w_i$.

In the chart generation step the algorithm considers all strings of length > 1. We will write w_{ij} to refer to the substring of S from word w_i to w_j. We will add a nonterminal N to $\mathcal{A}(i, j)$ given there is at least one rule $N \rightarrow A\ B$ and there exists a k, $1 \leq k < j$, such that A derives the first k symbols of w_{ij} and B derives the final $j - k$ symbols. This condition holds if and only if N can derive w_{ij} through one or more steps, written as $N \stackrel{*}{\Rightarrow} w_{ij}$.

3 Data and Implementation

The first module implemented is a grammar creator that extracts a PCFG from a given treebank corpus. The second is a CYK parser that creates a parse forest for a sentence given a PCFG in the format created by the first module. All code is implemented in Python.[1] We will first discuss the data used and then the implementation details of these modules.

3.1 The Penn Treebank

The implementation is designed to be used with the Penn Treebank Wall Street Journal (WSJ) corpus. For convenience we use a binarized version of this corpus in which all trees are binary trees, excepting trees headed by the start symbol TOP which may be unary or binary. The trees are presented in a text file as sentences bracketed with parentheses and tagged with the standard Penn treebank part-of-speech (POS) tags.

3.2 Extracting a PCFG from the Treebank

We chose the data structures of our PCFG such that we can quickly retrieve the left hand side (LHS) of any rule given the right hand side (RHS) and quickly retrieve the RHS given the LHS. Recalling N^i is a nonterminal and ζ^j is a sequence of terminals and nonterminals, the treebank is parsed into three data structures:

- `cfg_l2r`, a mapping from each LHS N^i to a set of RHSs $\{\zeta^j\}$.
- `cfg_r2l`, a mapping from each RHS ζ^j to a set of LHS terminals $\{N^i\}$.
- `pcfg`, a mapping from a tuple of LHS N^i and RHS ζ^j to $P(N^j \rightarrow \zeta^j | N^i)$.

We will occasionally use Python terminology and refer to these data structures as *dictionaries*.

To efficiently create these data structures we begin by creating `cfg_l2r` as a multi-set were the ζ^j need not be unique, noting that the cardinality of this multi-set will be equal to the total number of mappings from N^i to any ζ^j, which is the denominator in Eq. 2. Additionally, we calculate the numerator

[1] Code must be run under Python version 2.7 with the `argparse` module installed.

of Eq. 2 by incrementing $\texttt{pcfg}(N^i, \zeta^j)$ every time the production $N^i \to \zeta^j$ is seen, thereby giving us the frequency, $\mathcal{F}_{\mathcal{R}}(N^i \to \zeta^j)$. We finally convert `pcfg` to a probability distribution by setting $\texttt{pcfg}(N^i, \zeta^j)$ equal to itself, the relative frequency, divided by the cardinality of $\texttt{cfg\_l2r}(N^i)$, the total frequency for nonterminal N^i:

$$\texttt{pcfg}(N^i, \zeta^j) = \frac{\texttt{pcfg}(N^i, \zeta^j)}{|\texttt{cfg\_l2r}(N^i)|} \tag{4}$$

this is a normalization step. We then convert `cfg_l2r` from a multi-set to a set.

The actual parsing of the treebank is done by counting parentheses, which was found to be simpler and less computationally intensive than a recursive method. We check the correctness of the probability distributions in our PCFG by testing that the sum over all $N^i \to \zeta^j$ is equal to unity, as required by Eq. 1. Due to rounding errors, the largest divergence from unity is 0.00000000000022881697, which we do not expect to significantly affect our calculations. We also checked the correctness of our implementation by examining the productions created by hand.

3.3 CYK Implementation

Using the data structures described in Sec. 3.2 we can now use the CYK algorithm to generated parses of a sentence given our grammar. To efficiently keep track of which RHS productions cover the sentence, e.g. which ζ exist such that the production $TOP \to \zeta$ can generate a parse of the entire sentence, we store ζ in a dictionary `covering` indexed by i, j, TOP. To retrieve the set of productions from TOP covering the sentence we simply retrieve the entry $\texttt{covering}(0, n, TOP)$.

The chart initialization step is shown in Algorithm 1. After completion we will have filled the chart with all nonterminals that generate strings of length 1. After this is done we proceed to chart generation.

Algorithm 1 Chart initialization pseudo code.

Require: $\mathcal{C}$, `cfg_r2l`
  for $0 \leq i < n$ **do**
    for $A \in \texttt{cfg\_r2l}(w_i)$ **do**
      $\mathcal{C}(i, i+1) \leftarrow \langle A \rangle$
    end for
  end for

The chart generation step is shown in Algorithm 2. We represent i by *begin*, j by *span*, and k by *split* to accord with their semantic interpretation. The chart generation algorithm operates by considering all sentence spans of length 2 through n and for each span considering subsentences starting at 0 through $n - span$ and all splits within this span. For example, in the first iteration, $span = 2$, $begin = 0$, $end = 2$, and $split = 1$. On lines 5-6 the algorithm retrieves

the values of $\mathcal{C}(0,1)$ and $\mathcal{C}(1,2)$. Referring to Algorithm 1 we can see that these

Algorithm 2 Chart generation pseudo code.

```
Require: C, covering, cfg_r2l
 1: for 2 ≤ span ≤ n do
 2:    for 0 ≤ begin ≤ n − span do
 3:       end ← begin + span
 4:       for begin + 1 ≤ split ≤ end − 1 do
 5:          {A} ← C(begin, split)
 6:          {B} ← C(split, end)
 7:          for A_x ∈ {A} do
 8:             for B_y ∈ {B} do
 9:                for N ∈ cfg_r2l(A_x, B_y) do
10:                   C(begin, end).add(⟨N⟩)
11:                   covering(begin, end, N).add(⟨A, B⟩)
12:                end for
13:             end for
14:          end for
15:          check_unary(begin, end)
16:       end for
17:    end for
18: end for
```

values will be non-empty nonterminals that generated single word terminals. The algorithm will then proceed, in lines 7-9, to iterate over any rules that can be generated from the combination of the rules generating these terminals. As *begin* is incremented this same procedure continues for all possible beginnings and splits for strings of length 2, and then so on up to strings of length n.

Algorithm 2 only accounts for binary productions. Therefore, we see on line 15 that, after cycling over all the possible binary combinations, we call the function `check_unary` to determine if we can add any unary productions. This is shown in Algorithm 3.

The `check_unary` algorithm tests all nonterminals B covering the entire span *begin* to *end* and updates $\mathcal{C}$ and `covering` if there is a production $TOP \rightarrow B$. Our data set only has non-binary production from our start symbol TOP and therefore we will only consider TOP as a possible LHS. For other data sets with other non-binary rules the algorithm could be simply extended by referring to the `cfg_r2l` dictionary.

4 Results and Discussion

We applied our grammar generation module to the WSJ corpus generating the data structures describe in Sec. 3.2. We then ran experiments on this grammar to query various POS-tag statistics. Additionally, we applied our CYK algorithm to the grammar and a set of test sentences, creating a list of covering productions.

Algorithm 3 Pseudo code for a function to check for unary productions. We are only concerned with unary productions that have an LHS equal to our start symbol TOP.

Require: $\mathcal{C}$, `covering`, `pcfg`
Require: $begin, end$
1: $added \leftarrow true$
2: **while** $added$ **do**
3: $added \leftarrow false$
4: **for** $B \in \mathcal{C}(begin, end)$ **do**
5: **if** `pcfg`$(TOP, (B)) > 0$ **then**
6: $\mathcal{C}(begin, end)$.add(TOP)
7: `covering`$(begin, end, TOP)$.add$(\langle B \rangle)$
8: **end if**
9: **end for**
10: **end while**

4.1 Syntactically Ambiguous Words

We can determine the set of syntactically ambiguous words by retrieving all the values from the structure `cfg_r2l` which have a length greater than 1, indicating multiple nonterminals map to this particular terminal. In Table 1 we present four syntactically ambiguous words. Note that we did not performing any case folding, because case arguably provides significant information for a parser. In Step 3 we will examine case folded versus not case folded parses and evaluate this hypothesis to determine if case folding increases parse accuracy.

Table 1: Syntactically ambiguous words. Three significant digits have been retained.

Word	Possible Nonterminal	Probability of Production
no	DT	0.00740
	UH	0.0722
	RB	0.00271
Western	JJ	0.00108
	NNP	0.000875
lock	VB	26.5×10^{-5}
	VBP	8.01×10^{-5}
	NN	3.76×10^{-5}
	JJ	1.63×10^{-5}
refunding	VBG	3.01×10^{-5}
	NN	26.9×10^{-5}
	JJ	4.90×10^{-5}

These ambiguities are as expected. For example, the ambiguities for “no” are divided amongst determiner, interjection, and adverb. The highest probability

is assigned to interjection, which is significantly higher that the probability assigned to the next highest, determiner, and the lowest, adverb. This is expected and reflects our intuitive knowledge that "no" is most commonly used as an interjection in English.

4.2 Most Likely Productions

We can calculate the most likely productions for a nonterminal by retrieving the value $\texttt{pcfg}(N^i, \zeta^j)$, for each j such that there is a production $N^i \rightarrow \zeta^j$, and then ordering these productions by decreasing probability. We present the most likely production for the nonterminals $\{VP, S, NP, SBAR, PP\}$ in Table 2. The low

Table 2: Most likely productions for $\{VP, S, NP, SBAR, PP\}$. Three significant digits have been retained.

Nonterminal	Most Likely Production	Probability of Production
VP	$(VMD, VP@)$	0.0768
S	$(NP, S@)$	0.354
NP	$(DT, NP@)$	0.128
$SBAR$	(IN, S)	0.454
PP	(IN, NP)	0.796

probabilities of the most likely productions for VP and NP compared to the other nonterminals are expected because many more productions are headed by VP and NP, thus dispersing their probability mass to a larger degree.

4.3 Covering Productions

The covering productions can be generated for an arbitrary input sentence given the grammar. For example, consider the sentence:

$$\text{The finger-pointing has already begun .} \tag{5}$$

We will show the set of productions from our start symbol to nonterminals that can then generate this sentence, i.e. the set of nonterminals XP, YP such that $TOP \rightarrow XP$ (a unary production) or $TOP \rightarrow XP\ YP$ (a binary production) generate the sentence. This sentence is only covered by unary rules, which are given below:

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

where the sequence %%%% marks a binarization. Importantly, we see that S is present in this set, which is the correct production for this sentence as given by the gold standard parse tree for this test sentence.

5 Conclusion

In this report we have presented an effective way to extract a PCFG from a bracketed corpus of sentences. Our implementation stores additional data structures allowing quick access to production LHSs given an RHS and to production RHSs given an LHS. Making using of these structures we implement the chart parsing CYK algorithm, which generates a parse forest for a sentence based on an extracted grammar.

We presented results of a statistical analysis of the grammar, including syntactically ambiguous terminals and the most likely productions for common nonterminals. We presented an example covering production output of CYK applied to a test sentence used for the WSJ corpus. We found that the covering productions included the correct production $TOP \rightarrow S$.

We have built the fundamental mechanics needed for implementing Viterbi parsing, the inside algorithm and further extensions. It will be interesting to test the effectiveness of Viterbi parsing and determine what changes can be made to improve performance.

References

1. Daniel Jurafsky and James H. Martin, *Speech and language processing: An introduction to natural language processing, computational linguistics and speech recognition*, 1 ed., Prentice Hall, 2000.
2. Christopher D. Manning and Hinrich Schütze, *Foundations of statistical natural language processing*, MIT Press, Cambridge, MA, 1999.