Elements of Language Processing and Learning
Probabilistic Context-Free Parsing

R. Tobi (0448710)
J. van Turnhout (0312649)
N. Raiman (0336696)
{rtobi, jturnhou, nraiman}@science.uva.nl

October 30, 2012

Abstract

We describe our implementation and evaluation of a probabilistic context-free parser for natural-language sentences built for the course *Elements of Language Processing and Learning*. The parser is trained and tested on sections of the *Wall Street Journal* (WSJ) treebank.

1 Introduction

In the field of natural language processing (NLP) there are many theories on how to apply context-free grammars (CFG) in order to determine whether a sentence is grammatically correct. The reason for this is that there exist many exceptions under which plain CFGs perform poorly and analysing the syntactic structure of the sentence is not sufficient to determine grammaticality. A major common factor of these exceptions is ambiguity, and CFGs need augmentation to deal with this appropriately. Probabilistic context-free grammars (PCFG) are grammars with probabilities attached to each production rule in the grammar. These probabilities can be learned from treebank corpora and provide a solution for resolving ambiguity in sentences. In addition, the Cocke-Younger-Kasami (CYK) algorithm is an efficient algorithm for parsing sentences based on dynamic programming, and can easily be extended to probabilistic form to handle PCFGs. In this report we will discuss how a PCFG is extracted from a treebank, provide details on the CYK algorithm, and describe how its probabilistic extension to *P*CYK is implemented. Also, the importance of smoothing techniques will be addressed as they boost the success rate of the PCYK algorithm. The remainder of this text is organized as follows. In section 2, related work on PCFG, CYK and Good-Turing smoothing will be addressed. The implementation of the PCYK algorithm and data structures used for trees and PCFG are covered in section 3. We discuss our experiments and their results in section 4. Finally, we conclude in section 5.

2 Background

The PCFG used by the PCYK parser is a CFG with probabilities given to the production rules. These probabilities represent how probable a production rule is in the language and in our case are learned from the WSJ corpus, a collection of 40.000 POS-tagged sentences in binarized tree-structure format. For (P)CYK to be able to take a (P)CFG as input, the grammar must be in Chomsky Normal Form (CNF). A CFG is in CNF if it is ϵ-free and every production rule is of the form $A \Rightarrow BC$ or $A \Rightarrow a$, where A, B, and C are non-terminal symbols (eg. the POS-tags) and a is a terminal symbol (eg. an individual word). CNF trees are binary except for the pre-lexical nodes.

Unfortunately, the WSJ corpus violates some CFG rules: 1) most of the trees contain unary rules $A \Rightarrow B$ (eg. `NP` $\Rightarrow$ `N`); 2) the sets of non-terminals and terminals are not disjoint; 3) no unique start-symbol exists, but rather several special unary rules such as `TOP` $\Rightarrow$ `S` and `TOP` $\Rightarrow$ `SINV` of which the right-hand-sides are part of a set of "bonafide" start-symbols that indicate sentence type. Preprocessing steps are needed to deal with these violations. We solve 1) by converting the trees to CNF. To get around 2) we prefix each non-terminal tag by the string `CAT_`, so that eg. the tag 'CD' and the word 'CD' are distinct. Punctuation tags are treated similary. Lastly, 3) is handled by marking each of the right-hand-sides of `TOP` as a start-symbol in the grammar; these are not disjoint from the other non-terminals. When the PCYK algorithm parses a sentence successfully, it will have produced a tabular forest of tree structures. This forest (non-explicitly) stores at most as many complete parse-trees as there are start-symbols; in other words, the root-node tag of each of these indirectly present trees – all of which have their own probability – is taken from the start-symbol set. The best parse is then the tree from the forest whose probability is highest.

A property of the CYK algorithm is that its worst-case complexity is $O(n^3)$ where n is the number of words in the sentence being parsed. Because of this, the algorithm is most suited to parsing short[1] sentences. However, CYK guarantees to find a parse for a sentence if there is one, and in combination with a PCFG we can calculate

$$T^*(S) = \underset{T \in \tau(S)}{\operatorname{argmax}} P(T)$$

where $T^*(S)$ is the most probable parse-tree for a given sentence S. Note that because the PCFG is extracted from CNF trees, the output from our parser will also be a tree in CNF. The ground-truth we use to evaluate the PCYK parser is however not in CNF, so we require a conversion step to debinarize the output trees. Also, because of the probabilistic approach and because the parser will encounter words that are not in the training-data, smoothing is required to assign some non-zero probability to unseen incoming words. For this we apply Good-Turing to re-estimate the amount of probability mass for the lower-frequency frequencies.

[1]We consider any sentence of at most 15 words to be "short".

3 Implementation

The first step of the implementation is to learn the PCFG from the corpus. This is done by extracting the relative frequencies of the production rules from the tree-structure of the sentences and storing them in a frequency dictionary (associative array). The production rules are grouped by the non-terminals on the left-hand-side of the rules. Then the relative frequencies per rule are calculated by dividing its frequency by the total frequency of the left-hand-side non-terminal it belongs too. These are stored in a relative-frequency dictionary. By doing this the sum of probabilities of every unique left-hand-side non-terminal, for example `CAT_NP`, is equal to 1. In order to make this probabilistic parser robust against parsing sentences in which one or more words have not been encountered before, a prior probability must be calculated for *every possible* unseen word. This is done via Good-Turing smoothing. For this to work, we make the assumption that the probability of encountering a word with zero frequency is equal to encountering a word with frequency one in the production rules. Custom lexical production rules are added to the learned PCFG (e.g. `CAT_NP` $\Rightarrow$ `xxxunknown`), which make it possible to replace any unknown word with the placeholder `xxxunknown` so the sentence can be parsed. An extention to smoothing in our implementation allows letting this placeholder contain features like capital letters, hyphens or a suffixes if they are present in the real unknown word itself.

The feature-based rules are weighted according to the frequency of the left-hand-side symbols of terminal rules with the respective features occurring in them. For example, if capital letters occur for 50% in `CAT_NN` rules, 30% in `CAT_JJ` rules and the remaining 20% in other rules, the capital-based rule with left-hand-side `CAT_NN` will receive an additional weight of 0.5 during smoothing, `CAT_JJ` an extra weight of 0.3 and so on. At this point, the pre-calculations and processing are done and the PCFG is ready to be handed to the PCYK parser. We implemented the parser near-verbatim according to the description in

[1], see algorithm 1 for details[2]. Smoothing is done at line 13 in the algorithm. The parse-trees of a single sentence are constructed from the returned 3D table τ. For each of the start-symbols marked previously, we check if there is a span A starting at $i = 1$ of length $j = n$ in the history τ for that symbol, where n is the number of words (ie., a rule $A \Rightarrow BC$ covering all words in the sentence). If so, then we recursively visit its two sub-spans B and C, whose start- and end-indices (and hence their locations in the table) can be determined from i, j, and the span's split-position k recorded in τ, down to the pre-lexical level. Tree nodes corresponding to the span parent/child relations are inserted along the way. The tree whose start-symbol (root-node) is associated with the highest-probability parse is selected for evaluation.

4 Results

Evaluation is done using a provided evaluation program, a separate corpus of test-sentences, and a set of gold-standard trees (the ground-truth) aligned with these. Because PCYK returns a CNF tree, we need to debinarize each parse to have it match the structure of the gold-trees. This is a simple procedure in which we remove the binary node indicators (indicated by '@' characters in the training-set) and set the parent pointer for each of the children of a binary node N to N's parent. Thus, the parent P of N inherits N's children. We skipped each test-sentence longer than 15 words; taking care to align the trees in the ground-truth with the remaining 603 (out of a total of 2416) sentences. The grammar rules and probabilities were trained on all $\approx$ 40.000 sentences from the WSJ corpus. Note that our test-sentences are from a different section than (but from the same domain as) the training-data. We have tested three different setups for our experiments: PCYK without smoothing, PCYK with smoothing for unknown incoming words, and PCYK with smoothing for unknown incoming words based on features like hyphens, capital starting letters or suffixes. We measure in terms of pre-

Algorithm 1 PCYK parsing algoritm

```
 1: function CYKParse($G$, $N$, $S$, $P$)
 2: $G$ = grammar
 3: $N$ = list of non-terminal symbols
 4: $S$ = sentence
 5: $P$ = corpus-trained production probabilities
 6: $\pi$ = table of parsed-production probabilities
 7: $\tau$ = table of parsed-production spans
 8: for $w \epsilon S$ do
 9:    for $A \epsilon N$ do
10:       if $(A \Rightarrow w_i) \epsilon G$ then
11:          $\pi[i, 1, A] \leftarrow P(A \Rightarrow w_i)$
12:       else
13:          $unknown \leftarrow$ GT-smooth($w_i$)
14:          $\pi[i, 1, A] \leftarrow P(A \Rightarrow unknown)$
15:       end if
16:    end for
17: end for
18: for $j$ from 2 to len(S) do
19:    for $i$ from 1 to len(S)-$j + 1$ do
20:       for $k$ from 1 to $(j - 1)$ do
21:          for $(A \Rightarrow BC)$ in $G$ do
22:             $pB \leftarrow \pi[i][k][B]$
23:             $pC \leftarrow \pi[i + k][j - k][C]$
24:             $p \leftarrow (pB \cdot pC) \cdot P(A \Rightarrow BC)$
25:             if $p > \pi[i][j][A]$ then
26:                $\pi[i][j][A] \leftarrow p$
27:                $\tau[i][j][A] \leftarrow < k, B, C >$
28:             end if
29:          end for
30:       end for
31:    end for
32: end for
33: return $\pi, \tau$
```

[2]Note that this formulation traverses the Viterbi path implicitly.

cision, recall, f-measure (in which both precision and recall are weighted equally, ie. $\beta = 1$), complete match and the overall tagging accuracy.

Bracketing Recall	62.20
Bracketing Precision	68.28
Bracketing FMeasure	65.10
Complete match	13.76
Tagging accuracy	68.43

Table 1: Non-smoothed results.

Table 1 displays results from the non-smoothing experiment. Because the parser fails whenever an unknown word occurs within a sentence, we can expect huge improvement when we use smoothing techniques. In table 2, results for smoothing without features are displayed. As explained in section 3, the probabilities of the added production rules are the Good-Turing probabilities for N_1 (the group of words with occurrence frequency 1). Also, the probabilities of all production rules with frequency 1 are replaced with the Good-Turing probability for N_1.

	N_1
Bracketing Recall	69.91
Bracketing Precision	68.95
Bracketing FMeasure	69.43
Complete match	17.91
Tagging accuracy	82.67

Table 2: Results of smoothing *without* features.

In table 3, results for smoothing with features are displayed. Here we also varied the frequency of each production rule, and replaced its data-derived relative frequency by the relative frequency calculated with Good-Turing smoothing. The best f-measure is obtained when replacing the relative frequency estimates for production rules with frequency 1 to 10 (boundary N_{10}).

	N_1	N_{1-5}	N_{1-10}
Bracketing Recall	70.42	70.80	71.16
Bracketing Precision	69.46	69.76	70.28
Bracketing FMeasure	69.94	70.28	70.72
Complete match	19.40	19.40	22.22
Tagging accuracy	83.14	83.25	83.06

Table 3: Results of smoothing *with* features.

5 Conclusion

For the Good-Turing method, it is of great importance to find a boundary for which smoothing is effective. Our results show us that less smoothing can give a better tagging accuracy, but will cause the f-measure to suffer. When the smoothing boundary is set to frequency 10, the f-measure goes up, but the tagging accuracy slightly decreases. In theory we would like to maximize the bracketing f-measure because precision and recall are both of equal importance, and this represents the correctness of the tree-structure. In practice however, the tagging accuracy itself is also important since there are many NLP applications that have need of the correct POS-tag alone, but not the complete (sub)tree structure underneath it. Although PCFGs can address syntactic ambiguity, they are not without drawbacks. For example, the context-free assumption makes it hard to model long-distance dependencies between words. Future work on the subject of extracting a PCFG from the WSJ could be to perform parent-child annotation. By labeling every node of a tree in the training-data with the non-terminal symbol of its parent, the context-free assumption can be weakened.

References

[1] D. Jurafsky & J. Martin:
Speech & Language Processing
(International Edition, 2000)

[2] G. Kochanski:
Good-Turing probability estimation
(http://kochanski.org/gpk/teaching/0401Oxford/GoodTuring.pdf)