PROBABILISTIC CONTEXT-FREE GRAMMARS IN NATURAL LANGUAGE PROCESSING

S. Sundararajan

Center of Advanced Vehicular System
Mississippi State University
srinivas@cavs.msstate.edu

ABSTRACT

Context-free grammars (CFGs) are a class of formal grammars that have found numerous applications in modeling computer languages. A probabilistic form of CFG, the probabilistic CFG (PCFG), has also been successfully applied to model natural languages. In this paper, we discuss the use of PCFGs in natural language modeling. We develop PCFGs as a natural extension of the CFGs and explain one probabilistic parser for PCFGs in detail. We also outline two methods that are used for estimating the PCFG probabilities. Finally, we state the limitations of PCFGs and mention how it can be augmented for modeling natural languages better.

1. INTRODUCTION

The field of natural language processing (NLP) is primarily concerned with the understanding of the structure (syntax) and the meaning (semantics and pragmatics) of spoken and written natural languages [1]. To make a computer understand a natural language, it is first necessary to define a language formally, but natural languages are much too complex for representation by any formal language. By making simplifying assumptions, we can approximate natural languages by formal languages. Better the approximation we need, less relaxed the assumptions we can make and more complex the formal language model becomes.

Of the different types of formal grammars, one that gives the best trade-off between complexity and degree of approximation is the context-free grammar (CFG). By itself, the power of CFGs is very limited due to the problem of ambiguity [1][4][5], i.e., sentences in a natural language are frequently ambiguous which CFGs cannot handle. To disambiguate such sentences, CFGs should be extended to work in a probabilistic setting, leading to probabilistic context-free grammars (PCFGs) [1][4][5]. There has been a plethora of research to extend parsing algorithms for CFGs, like Earley and Cocke-Younger-Kasami (CYK) algorithms [1, 3], to work with PCFGs. Several procedures to estimate PCFG probabilities like inside-outside algorithm have been proposed and successfully applied in practice.

Unfortunately, PCFGs are much too simple to model the sentences from a natural language. To address this inadequacy, several extensions [1][4]] to PCFG model have been invented. While some languages including English can be modeled by some extended form of PCFGs, there are others for which PCFGs are totally incapable of modeling and for which we need to resort to models that are more stringent in their assumptions they make.

The paper is organized as follows. In section 2, the concept of formal grammar and language is introduced in a general setting and the types of languages are listed with a brief description of each. Section 3 deals with CFGs in particular. An example CFG is used to demonstrate the problem of ambiguity. PCFGs are introduced in section 4 as an aid to disambiguate ambiguous sentences. One algorithm for parsing, called Cocke-Younger-Kasami algorithm is explained in detail, in a probabilistic setting. Methods to learn the production rule probabilities are also outlined. In section 5, a summary of the drawbacks of PCFGs is given along with a brief description of extensions to PCFGs to alleviate some of its problems.

2. FORMAL LANGUAGES AND GRAMMARS

A set of strings of symbols constitutes a language [1][2][3][4]. To define a language, we can enumerate all the valid strings in the set, but it is more useful to specify a language using a generative grammar or a set of production rules (P). There are two types of symbols that are used for writing production rules. Those symbols that appear directly in the language are called the terminal symbols while the non-terminal symbols appear only as intermediate constructs in the derivations of terminal symbols. For example, consider a “noun phrase” (NP) of an English sentence “that book”. The symbols “that” and “book” occur in the language and, hence, are terminal symbols. On the other hand, the symbol NP is a non-terminal symbol because it refers to a constituency of the two words and by itself it is not in the language. One special non-terminal symbol is the start symbol, S, which represents a sentence in a language.

A commonly used general form of a production rule is [3]:

$$lAr \rightarrow lBr \quad (1)$$

In this scheme, only one non-terminal, *A*, on the left side of the rule, is allowed to be replaced by a string, *B*, of terminals and/or non-terminals. The non-terminals *l* and *r* on either side of A, form the left and right contexts respectively, and are left unmodified when moving from left to right in the rule.

2.1 **Types of Grammars**

Based on the forms of the production rules, grammars are classified into four types [3], forming a hierarchy called Chomsky hierarchy:

a) Type 0 grammars have rules that are unrestricted forms of Eq. 1. Hence they are also called as unrestricted grammars. Languages based on type 0 grammars are called recursive languages. These grammars are equivalent to a class of machines called Turing machines.

b) Type 1 grammars are similar to type 0 ones except that the deletion of a non-terminal is not allowed. Such grammars are also called context-sensitive grammars. The equivalent machine structures that can recognize all the strings and only the strings produced from this grammar are the linear bounded automata.

c) Type 2 grammars have rules that do not have any context. This can be expressed in the Chomsky normal form (CNF) as:

$$A \rightarrow BC \mid a \quad (2)$$

Here *A*, *B* and *C* are non-terminals while *a* is a terminal symbol. Due to the absence of context, these grammars are called context-free grammars (CFGs). Strings from a language generated from any CFG can be recognized by a machine called push-down automaton.

d) Type 3 grammars have type 2 rules, with the additional restriction that replacement strings be either a single terminal symbol or a single terminal symbol followed by a non-terminal symbol.

$$A \rightarrow xB \quad (3)$$

Here *x* is a terminal symbol while *B* is a non-terminal symbol, which may be an empty string. Regular languages are equivalent to finite-state machines (FSMs).

3. CONTEXT-FREE GRAMMARS

It is well known that most natural languages including English may not be context-free [1]. Yet, CFGs are very commonly applied in natural language modeling because context-sensitive grammars are very complex to deal with. Moreover, CFGs can be extended to augment some of the dependency issues. When applied to linguistics, CFGs are also called phrase-structure grammars. The reason for this terminology stems from the fact that the non-terminals of the CFG capture the grammatical structure in the language by dividing the sentences into phrases. For example, CFG applied to sentences in the English language would typically include non-terminals like noun-phrase (NP) and verb phrase (VP). The intuition behind this is the idea of constituency – a group of words (a phrase) behaving as a single unit (like noun, verb, etc.).

A CFG can be defined formally as:

$$CFG = (N, \Sigma, P, S)$$

$$N \rightarrow \text{a set of non-terminals}$$

$$\Sigma \rightarrow \text{a set of terminal symols}$$

$$P \rightarrow \text{a set of production rules in CNF}$$

$$S \rightarrow \text{start symbol}$$

A CFG, like any other grammar, can be used in two ways – as a generator for sentences in the language or as a parser for assigning structure to a given sentence. Our concern in this paper is to use CFG as a parser.

CFGs are conventionally written in Chomsky normal form (as in Eq. 2) or some variant of Backus-Naur form (BNF). Any CFG can be converted to CNF form and any grammar in CNF form is a CFG. When the CFG is specified in CNF, the derivation of a string of length *n* is always *2n-1* steps long, every parse tree is a binary tree and the height of the tree is at most the length of the string [2, 3]. These properties make CNF the most commonly used form for CFG representation.

There are several well-known dynamic programming-based parsers for CFG: the Cocke-Younger-Kasami (CYK) algorithm, the Earley algorithm and the Graham-Harrison-Ruzzo algorithm [1][2]. CYK algorithm is presented in detail in a probabilistic framework in the next section.

An example grammar, CFG1 that would be used for illustrating some of the ideas behind CFGs and parse trees is shown in Figure1. This example grammar can easily be made to represent a grammar from a natural language by modifying the rules, non-terminals and terminals appropriately.

Now consider a string xxz. One possible parse tree for this string is shown in Figure2. A little examination reveals that there is only one tree that parses that string. Thus the string xxz is unambiguous as for as this grammar is concerned. This is not true always. If for instance, we consider another string xyz, we find that there are two possible parse trees as shown in Figure 3. The ambiguity of which tree represents the correct parse cannot be resolved unless we extend the concept of CFGs in a probabilistic framework.

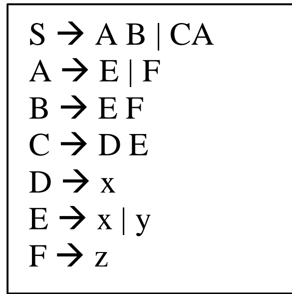


Figure 1: An example CFG grammar, CFG1

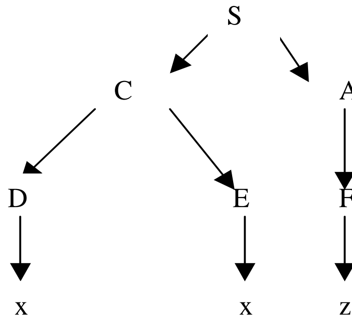
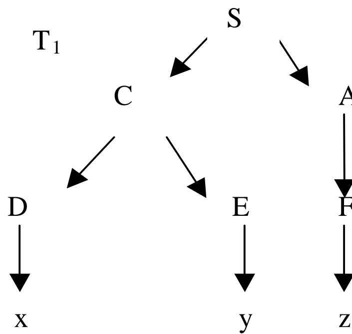


Figure 2: Parse tree for string xxz with CFG1



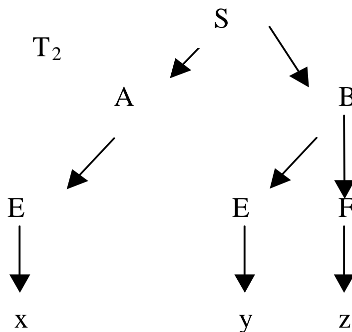


Figure 3: Parse trees for string xyz with CFG1

The reason behind the success of CFG and its variants in competing with other models like hidden Markov models (HMMs) and N-grams, in natural language modeling, is due to the occurrence of non-local dependencies [5]. While HMMs and N-grams can do a good job of modeling local dependencies, they cannot model the dependencies between parts of a sentence that are separated far apart. CFGs can complement the other models in this aspect.

4. PROBABILISTIC CONTEXT-FREE GRAMMARS

The most commonly used type of grammar in natural language modeling is a probabilistic version of the CFG, called probabilistic (or stochastic) context-free grammar (PCFG). A PCFG is a 5-tuple [1][5]:

$$PCFG = (N, \Sigma, P, S, D)$$
$$N \rightarrow \text{a set of non - terminals}$$
$$\Sigma \rightarrow \text{a set of terminal symols}$$
$$P \rightarrow \text{a set of production rules in CNF}$$
$$S \rightarrow \text{start symbol}$$
$$D \rightarrow \text{function assigning probabilities to each rule in P}$$

The function *D* expresses the probability P that a given non-terminal *A* derives the sequence β.

The main reason for augmenting a CFG with probabilities for production rules is that PCFGs can be very useful in disambiguation [1][5]. Ambiguity arises when more than one parse-tree spans the input string and this phenomenon occurs commonly in natural language parsing. With our example CFG grammar, we found that two trees can parse the string xyz. In such cases, knowledge of probabilities concerning a sentence and its parse-trees can help choose the correct tree that is most probable, i.e., we choose:

$$\hat{T}(S) = \operatorname*{argmax}_{T \in \tau(S)} P(T \mid S) \tag{4}$$

where $\tau(S)$ is the set of all parse trees for S. Another advantage of PCFGs is that it can assign a probability to the string of words in a sentence, which can be very useful in language modeling.

Let us now try to disambiguate the string xyz with our example grammar. But to do this, we need to first augment CFG1 by assigning probabilities for each production rule and form PCFG1, defined as in Figure 4.

Given the two possible trees for the string *xyz*, the probability for each of them can be computed by multiplying together each of the rules used in the derivation.

$P(T_1) = 0.3 * 1.0 * 1.0 * 1.0 * 0.3 * 1.0 = 0.09$

$P(T_2) = 0.7 * 0.6 * 1.0 * 0.3 * 0.7 * 1.0 = 0.0882$

As T_1 has higher probability of occurrence than T_2, a disambiguation algorithm that selects the parse with the highest PCFG probability would choose T_1.

Rule	Probability
S → A B	0.7
S → C A	0.3
A → E	0.6
A → F	0.4
B → E F	1.0
C → D E	1.0
D → x	1.0
E → x	0.3
E → y	0.7
F → z	1.0

Figure 4: An example PCFG grammar, PCFG1

4.1. Probabilistic CYK parsing of PCFGs

CYK algorithm [1][2] is a bottom-up dynamic programming approach. It assumes that the input, output and data structures are in the following form:

Input:

- PCFG in the Chomsky Normal form, with m non-terminals having indices 1, 2, …, M. Start symbol S has index 1. It is assumed that the probabilities for all the rules, $P(A \to BC)$ is known.
- L words $w_1 \dots w_L$.

Data Structure: An array p(i, j, A) that holds the probability of the most probable tree that has non-terminal A as the root node and spans the input words from i to j. An array, back_ptr, is used for storing the back-pointers to links between the non-terminals in the parse tree.

Output: The maximum probability of the parse will be in p(1, L, 1), as this is the maximum probability among all trees starting at non-terminal with index 1, i.e. S, and spanning all the words in the sentence. The set of back-pointers will be in the matrix back_ptr from which the maximum likelihood tree can be constructed.

The algorithm as shown in Figure 5, proceeds in two stages. Here, w_{ij} is used to represent the sequence of words from w_i to w_j.

- Base case: In this all the rules of the form $A \to w_i$ are considered for each word in the input sentence and for each non-terminal A. The corresponding probability is stored in p(i, i, A).
- Recursive case: For every subsequence of the input string, w_{ij} and for every non-terminal, A, A derives w_{ij} if and only if there is a rule $A \to BC$ and a k, $1 \le k < j$, such that B derives w_{ik} and C derives w_{kj}. k acts as the splitting point that splits the word sequence w_{ij} into w_{ik} and w_{kj}. For every non-terminal A, the inner loops consider all possible splitting points and all possible non-terminals for B and C. When there is more than one such rule that can parse w_{ij}, the maximum probability of all such parses is stored in p(i, j, A), while back_ptr(i, j, A) stores the corresponding k, B and C. Once all the loops are executed, the most probable parse tree can be built by going through the set of back-pointers starting from the S node and going down the tree.

L : length of input string
M : number of non-terminals in grammar G

- Create and clear p(L,L,M) and back_ptr(L,L,M).
- Base case

```
for i=1:L
  for A=1:M
    if A -> w_i is in G
      p(i, i, A) = P(A -> w_i) ;
```

- Recursive case

```
for span = 2:L
  for start=1:L-span+1
    end=start+end-1;
    for k=start:end-1
      for A=1:M
      for B=1:M
      for C=1:M
        prob=p(start,k,B)*p(k+1,end,C)* P(A -> BC)
        if (prob > p(start, end, A))
          p(start,end,A)=prob;
          back_ptr(start,end,A)={k, B, C};
```

Figure 5: Probabilistic CYK algorithm (adapted from [1][2])

Informally stated, what this algorithm does can be summarized as follows:

Every substring of the input sequence is considered starting from substring length 1, then going on to substring length 2 and so on. For every substring of length *l* and for every non-terminal symbol, it finds all rules that span the substring for

each possible split point in the substring. It remembers only the rule with the maximum probability among all the rules and among all the split points. Once the whole length of the string is processed, all the remembered best paths (maximum probability) can be traversed through, to reconstruct the parse tree.

4.2 Learning PCFG probabilities

When laying out the structure of the input for CYK algorithm, it was assumed that the production rule probabilities, $P(\alpha \rightarrow \beta)$ are known. Now two methods to learn these probabilities are outlined [1][5].

There is a simple way to find the rule probabilities if we have a corpus called a treebank. This is a database of parse trees of sentences from a language. Given a treebank, estimating the rule probabilities becomes just counting and normalization.

$$P(\alpha \rightarrow \beta) = \text{Count}(\alpha \rightarrow \beta) / \text{Count}(\alpha) \tag{5}$$

When a treebank is not available, the counts for rule probabilities can be found by first parsing the corpus. This is simple for unambiguous sentences. In effect, we create a treebank by parsing each sentence in a database and storing the parse trees. Once we have this treebank, we can proceed in a similar fashion as that of the previous method. If the sentences are ambiguous and we need to find the PCFG probabilities, then we get into a deadlock: to find the probabilities we need to be able to parse ambiguous sentences but to parse ambiguous sentences we need to know the probabilities first. We can think of this as similar to missing data (PCFG probabilities) problem, and use the expectation-maximization (EM) algorithm to find the parse trees (E-step) and the PCFG probabilities (M-step) iteratively. In practice, a dynamic programming instance of the EM algorithm, called the Inside-Outside algorithm, is used for learning the probabilities.

5. LIMITATIONS AND EXTENSIONS OF PCFGS

PCFGs were introduced as an extension to CFGs to aid in sentence disambiguation, but they have a number of problems. Due to this, in practice, most current probabilistic parsers use some augmented form of PCFGs. The main drawback of PCFGs is that they do not model dependencies [1].

Although it was not stated explicitly, it is clear that the formulation of PCFGs assumes that the derivation from each non-terminal node to a set of input words, is not only independent of the nodes outside the sub-tree but also independent of the words on both sides of the subsequence of input string that the sub-tree considers. The first one refers to structural independence while the other implies lexical independence. Natural languages are not that simple and have both kinds of dependencies [1][4].

All extensions of PCFG try to include the dependencies between words and parse trees some way or the other. One drawback of extended PCFGs is that they need an extremely large corpus for estimating that probabilities. To avoid this, the various extensions consider some simplifying assumptions of independence. A commonly used solution to incorporate dependencies into PCFGs is the probabilistic lexicalized CFGs. This is based on the concept of head-driven grammars. Every phrase is associated with a "head" word, which constrains the overall structure of the sentence. Instead of computing the probability of the parse just by multiplying each of the PCFG rule probabilities, each rule probability is now conditioned on its head [1][4].

There are also some grammars, like dependency grammars [1][4], that make lexical dependency as the major constraining information, attaching no importance at all to constituents and phrase-structure rules. Such grammars are especially useful in modeling relatively free word order languages, i.e., languages in which words from the same sentence may be jumbled and still be semantically same. There is no way by which we can extend PCFGs to model these languages.

6. SUMMARY

This paper introduced first, the concept of formal languages and grammars. Some aspects of context-free grammars were discussed in detail. The problem of ambiguity that arises when parsing CFGs was explained with an example. Probabilistic context–free grammars were introduced as a means to achieve disambiguation. One probabilistic parser, the probabilistic CYK algorithm, was described in detail and two methods to learn PCFG probabilities were outlined. Finally the limitations of PCFGs in natural language processing were mentioned and some ways to extend PCFGs to address some of these limitations were presented.

7. REFERENCES

[1] Daniel Jurafsky and James H. Martin, *Speech and Language Processing,* Prentice Hall, NJ, 2000.

[2] A. V. Aho and J. D. Ullman, *The Theory of Parsing, Translation, and Compiling, Volume I: Parsing*, Prentice Hall, NJ, 1972.

[3] Richard Y. Kain, *Automata Theory: Machines and Languages*, McGraw-Hill, USA, 1972.

[4] Chris Manning and Hinrich Schutze, *Foundations of Statistical Natural Language Processing*, MIT Press, Cambridge, 1999.

[5] Detlef Prescher, "A tutorial on Expectation-Maximization algorithm including Maximum-Likelihood Estimation and EM training of Probabilistic Context-Free Grammars", *15th European Summer School in Logic, Language and Information*, University of Amsterdam, Amsterdam, 2003.