Technische Universität Dresden
Fakultät Informatik
Institut für Theoretische Informatik
Lehrstuhl Grundlagen der Programmierung

Bachelorarbeit

Training of probabilistic context-free grammars

Michael Jobst

Eingereicht am 28.06.2013
Bearbeitet vom 08.04.2013
bis zum 28.06.2013

Verantwortlicher Hochschullehrer:
Prof. Dr.-Ing. habil. Heiko Vogler
Betreuer:
Dr. Torsten Stüber, M. Sc.

Contents

1 Introduction

Ever since the appearance of science fiction movies, man has dreamt of machines, which, or better to be said who would act and think like humans. The field of artificial intelligence is one of the most important and wide spread topics in modern computer science. In the last years machines not only have gained the capability of answering to simple requests just like a pocket calculator, but also of automatic driving on public roads or offroad, of playing games like chess, even winning against world champions or of working with human language. Latter has only been achieved in a basic way up to now: Neither understanding nor translating nor speaking is working in general, though strongly limiting the expectations has already led to surprisingly good results.

Today, speech recognition, without understanding the contents, has become a well-functioning standard application. Also, machine translation of technical texts, like user manuals, leads to usable results. But we do not only want to evaluate the quality of a sentence, to translate without considering semantics or to produce text by using tricks like having a chat bot ask the user questions in order to be perceived empathic. What we are really looking for is to understand the contents of a text by structuring language into objects with a function and linking the meaning of sentences together in order to track objects through a text.

But how can we structure spoken language in order to make computers "understand" it someday? A language consists of sentences, which themselves can be split into words. Each of those words not only has a function, but also is related to a subset of the rest of the sentence. For modeling this, linguists use grammars, which describe the structure of a sentence and the function of its sub-sentences and words. Of course those grammars are books written for humans, which describe a language but somehow have to be structured so that they can be used by a program. Therefore several different approches exist, some simpler and less accurate, some more complex and powerful.

Taking a strongly inflecting language like czech, the potential complexity of such a grammar can quickly get enormous. Given a small dictionary of only 5000 words, we already have at least 5000 entries for sorting each word into a grammatical category. Furthermore for each noun there exist seven different declinations and the same amount for its plural forms; verbs need an entry for three different persons in singular and plural each, differing between male, female and neuter and so forth. We soon realize that our

initial amount of entries is far not enough. Now imagine a new entry for each grammatical relation between the words - needless to say the grammar will get very big, which makes manually writing it difficult and slows down the processing by a computer.

So it is better to keep things simple and especially to avoid overfitting[1], which however means that our grammar will "allow" all the possible sentences of a language, but also a lot of sentences which make no sense at all. Even worse, we can not even use it for recognizing the meaning of the words in a sentence as natural language is ambiguous. The english word "saw" could for example be the simple past of "to see", the tool used for cutting or the infinitve of the verb associated with the tool.

We could now think, that it is a good decicion to find a grammar, which completely avoids the misuse of a word by creating the rules in a way that there is only one grammatical interpretation for each sentence. But what about the sentence "I saw her duck." It can have three meanings: I could have seen her as she ducks, or I could have seen her animal or I could hurt it using a saw. As latter is not very nice it will probably be a less frequent sentence as the other ones. And the first sentence is also probably less likely to appear than the second.

Talking about frequencies and likelihood it is evident that the sentences and therefore the grammar rules somehow have different probabilities of occurance. But it is not possible to intuitively assign probabilities to the rules. However computer linguists have found training methods, which can relieve us of this task and basically work by evaluating large samples of language.

In this thesis, first a special type of such grammars is treated, called context-free grammars, then it will be extended using probabilities and two training methods are described. Furthermore, an implementation of those methods for Vanda, the modular machine translation system of the Dresden University of Technology, using Haskell, is presented.

[1] Overfitting describes that a model is too adapted to a specific sample (for example a book) to work for other samples, i.e. sentences which are not in the book. An extremely overfitted model would be a grammar which accepts only the sentences in this book. The contrary would be one, which accepts every sentence, no matter how senseless it is.

2 Context-free grammars

Looking for a grammar which is not that complicated but relatively powerful we come across *context-free grammars* (short: CFG) (Booth 1969; Booth and Thompson 1973). Like many grammars in formal language theory, context-free grammars basically are a set of replacement rules for variables. What makes them special is that the applicability of a rule is only decided by a single variable and not by its context. In the further sections, context-free grammars will be pictured and defined.

2.1. Definition

A context-free grammar is a 4-tupel $G := (\mathcal{V}, \mathcal{T}, S_0, \mathcal{R})$ with:

- an alphabet of variables $\mathcal{V}$
- an alphabet of terminals $\mathcal{T}$ with $\mathcal{V} \cap \mathcal{T} = \emptyset$
- a starting symbol $S_0 \in \mathcal{V}$
- a set of rules $\mathcal{R} \subseteq \mathcal{V} \times (\mathcal{V} \cup \mathcal{T})^*$

The variables are also called syntactic categories. $(\mathcal{V} \cup \mathcal{T})^*$ is the Kleene closure of the both alphabets' union. A word of the alphabet $\mathcal{T}$ is called *sentence* and a word of the alphabet $(\mathcal{V} \cup \mathcal{T})^*$ is called *sentential form.*

An example for a valid rule $R \in \mathcal{R}$ with $S, A, B \in \mathcal{V}$ and $c \in \mathcal{T}$ is:

$$R = (S, ABc)$$

For easier reading each rule $R \in \mathcal{R}$ can be written by separating the elements of the pair by a right-arrow, symbolizing an assignment to the variable on the left. Therefore our example rule R can also be written as:

$$R : S \to ABc$$

Following intuition, the first element of the pair R will also be called left(R) and the second element will be called right(R).

2.1.1. Application of rules and derivations

The application of a rule (a *derivation step*) is defined as a relation $\Rightarrow$ where $G = (\mathcal{V}, \mathcal{T}, S_0, \mathcal{R})$ is the grammar which is used. For making the application of more than one derivation step deterministic, we will only treat leftmost derivations, meaning that we can only replace the leftmost variable in a sentential form. Given a rule $R \in \mathcal{R}$, then

$$\overset{R}{\Longrightarrow} := \{(lVr, lsr) \mid l \in \mathcal{T}^*,\ s, r \in (\mathcal{V} \cup \mathcal{T})^*,\ \text{left}(R) = V \in \mathcal{V},\ \text{right}(R) = s\}$$

Let V be a variable and w be a sentence of G and d be a sequence of rules $R_0 \dots R_n$ with $V \overset{R_0}{\Longrightarrow} \dots \overset{R_n}{\Longrightarrow} w$. Then d is called the *derivation of w from V*. If $V = S_0$, d is called a *complete derivation*. The reflexive transitive closure of our relation ($\Rightarrow^*$) is also called the *derivation* relation.

$$\Rightarrow^* := \{(V, w) \mid V \in \mathcal{V}, w \in (\mathcal{V} \cup \mathcal{T})^*, \exists d = R_0 \dots R_n \in \mathcal{R}^* \mid V \overset{R_0}{\Longrightarrow} \dots \overset{R_n}{\Longrightarrow} w\}$$

We can also define the functions left and right for a derivation $d = R_0 \dots R_n$:

$$\text{left}(d) := \text{left}(R_0)$$
$$\text{right}(d) := w \text{ with } \text{left}(R_0) \overset{R_0}{\Longrightarrow} \dots \overset{R_n}{\Longrightarrow} w$$

The set of all complete derivations of a grammar G is called D_G. It is defined as follows:

$$D_G := \{d \mid \text{left}(d) = S_0, \text{right}(d) \in \mathcal{T}^*\}$$

2.1.2. The language of a grammar

Having defined derivations we can now think about the *language* a grammar produces. The language $\mathcal{L}_G \subseteq \mathcal{T}^*$ of a grammar G is the set of all sentences it can produce with complete derivations. $\mathcal{L}_G$ can be finite but does not necessarily have to. The definition of the language of a grammar strongly relates to the definition of complete derivations: If there exists a complete derivation from S_0 to w, w is a sentence of the language. Therefore:

$$\mathcal{L}_G = \{w \in \mathcal{T}^* \mid S_0 \Rightarrow^* w\}$$

2.2. An example grammar

As an example we will use the grammar $G = (\mathcal{V}, \mathcal{T}, S_0, \mathcal{R})$ which looks as follows:

- $\mathcal{V} = \{\text{DT, NNP, NN, NP, S, VP, VBD}\}$
- $S_0 = \text{S}$
- $\mathcal{T} = \{\text{a, man, John, Mary, saw}\}$
- $\mathcal{R}$ is the set of the following rules:

(1)	S $\rightarrow$ NP VP	(2)	NNP $\rightarrow$ Mary	(3)	NN $\rightarrow$ man
(4)	NP $\rightarrow$ DT NN	(5)	NNP $\rightarrow$ John	(6)	VP $\rightarrow$ VBD NP
(7)	NP $\rightarrow$ NNP	(8)	DT $\rightarrow$ a	(9)	VBD $\rightarrow$ saw

The names of the variables are syntactical categories as used in the Penn Treebank (Marcus, Santorini, and Marcinkiewicz 1993), for example NP stands for noun phrase. The symbols used in this work are described in Appendix A.

The rules have been numbered, so that for example $\overset{(1)}{\Longrightarrow}$ means a derivation step using rule (1).

At first glance it is not easy to see which language even such a simple grammer produces. Derivating a first sentence step by step can give us an idea. For easier reading the symbols are marked as follows: An underlined symbol gets replaced by the next application of a rule and the bold symbols are the result of the replacement of the last step.

$$
\begin{aligned}
\underline{\text{S}} &\overset{(1)}{\Longrightarrow} \mathbf{\underline{NP}\ VP} \\
&\overset{(4)}{\Longrightarrow} \mathbf{\underline{DT}\ NN}\ \text{VP} \\
&\overset{(8)}{\Longrightarrow} \mathbf{a}\ \underline{\text{NN}}\ \text{VP} \\
&\overset{(3)}{\Longrightarrow} \text{a}\ \mathbf{man}\ \underline{\text{VP}} \\
&\overset{(6)}{\Longrightarrow} \text{a man}\ \mathbf{\underline{VBD}\ NP} \\
&\overset{(9)}{\Longrightarrow} \text{a man}\ \mathbf{saw}\ \underline{\text{NP}} \\
&\overset{(7)}{\Longrightarrow} \text{a man saw}\ \mathbf{\underline{NNP}} \\
&\overset{(2)}{\Longrightarrow} \text{a man saw}\ \mathbf{Mary}
\end{aligned}
$$

2.3. Parse trees

Derivations can also be depicted as trees, starting with S_0 as root node and the nodes having the results of the application of the next rule as child nodes. In Figure 2.1 are the trees of some derivation steps of our example. The last one is a[1] *parse tree* of the sentence "a man saw Mary". It can also be written in a compact form:

[1] Using a more complex grammar it is very probable that there exist more parse trees of a sentence.

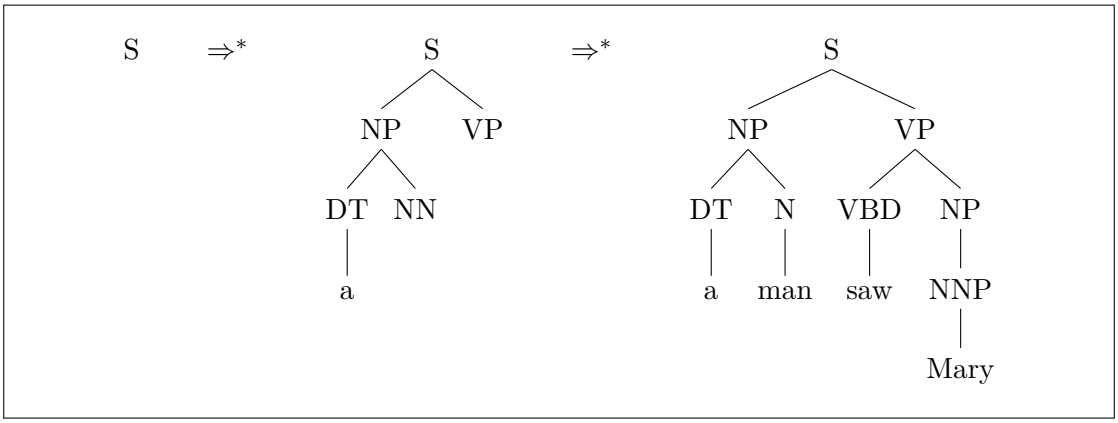


Figure 2.1.: Some steps of an example derivation of the sentence “a man saw Mary”

S (NP (DT (a) NN (man)) VP ((VBD saw) NP (NNP (Mary))))

Here an opening bracket depicts the beginning of a list of all child-nodes with their respective subtrees again written in brackets behind themselves.

From this structure, all the rules which produced the tree can be read off without knowing the trees of each derivation step. For example, looking at the root node of the parse tree, the rule (1) can be reconstructed by taking the node S as the left side and its direct children as the right side.

Reading off those rules can be done using the recursive function readoff.

Let $T(G)$ be the set of all possible parse trees of G. First, we need an auxiliary function $\text{head} : \mathcal{T} \to \mathcal{V} \cup \mathcal{T}$, which returns the first symbol of the tree.

$$\text{head}(t) = \begin{cases} V & \text{if } t = V(t_0 t_1 \dots t_n),\ V \in \mathcal{V} \text{ and } t, t_i \in T(G) \\ a & \text{if } t = a \text{ with } a \in \mathcal{T} \end{cases}$$

Now let $\mathcal{R}^*$ be the set of all possible sequences of rules in $\mathcal{R}$

$$\text{readoff}(t) = \begin{cases} \emptyset & \text{if } t = a \text{ with } a \in \mathcal{T} \\ X \to \mathit{head}(t_0) \dots \text{head}(t_n), \text{readoff}(t_0), \dots, \text{readoff}(t_n) & \\ \qquad \text{if } t = X(t_0 t_1 \dots t_n) \text{ and } t, t_i \in T(G) & \end{cases}$$

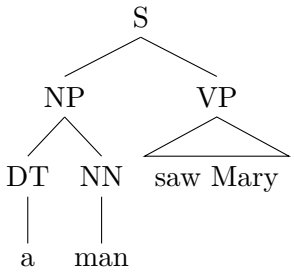
This sequence now can also be transformed into a complete derivation. Let $R_0, R_1, \dots, R_n$ be a sequence of rules from the application of *readoff* on a parse tree of a word w and S_0 be the starting symbol of the grammar which was used. Then the derivation d of the belonging parse tree is:

$$d = R_0 R_1 \dots R_n$$

As we have seen, parse trees can be transformed to derivations and vice versa, therefore in the future I will apply functions defined for parse trees to derivations and vice versa.

Though it is to be said, that a sentence can possibly have multiple derivations (and therefore multiple parse trees) which means that sentences can only be transformed to sets of parse trees or derivations whereas the best one has to be found. This is called parsing and will be treated later.

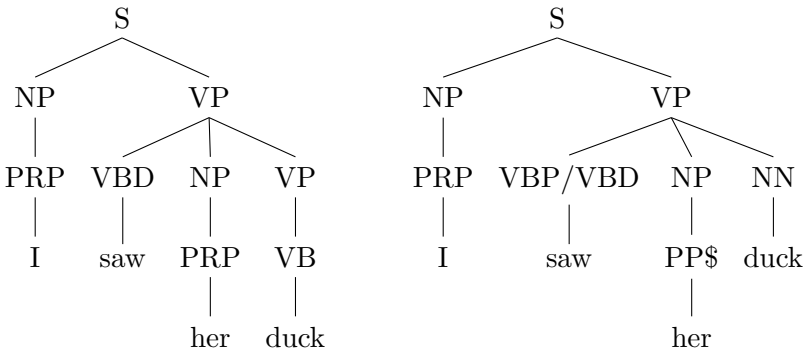
Parse trees or parts of them can also be abbreviated if we are just interested in a specific branch of the tree. For that computer linguists use a roof under which the part of the sentence generated by the branch is written. Such a tree can look like this:



3 Probabilistic context-free grammars

In *probabilistic context-free grammars* (Booth 1969; Booth and Thompson 1973) the rules are rated by probabilities. The reason for this approach is the assumption that each sentence is "generated" by its speaker or writer with a certain frequency. A sentence like "Hello!" is obviously used a lot more frequently than the one you are just reading.

Using probabilistic context-free grammars, it is also possible to disambiguate ambiguous sentences (Prescher 2005, 32-37). Our example grammar is not ambiguous, which means that each sentence has exactly one parse tree. For such a small and finite grammar this can be easily checked by looking at all parse trees. There are no two of them which have the same yield[1]. This however is not common in natural language: just think about our initial example, "I saw her duck." It could have one of the following parse trees:



The first one describes her action of ducking. In the second one, the two possible other meanings are included. The version with VBP is the usage of the tool and the one with VBD describes that she sees the bird.

Using probabilities we can decide, which analyze would be the better (meaning more frequent) one. In Prescher's tutorial (2005) some more complex examples for disambiguation can be found.

[1]The sentence which consists of all the terminals read from left. yield is defined in Section 4.2.1.

3.1. Modelling probability

For using probabilistic context-free grammars, we will first need some basic definitions for modelling probability.

3.1.1. Probability distributions

A *probability distribution* p on a set X is defined as:

$$p : X \to \mathbb{R}$$

p has to assign a value between 0 and 1 to each element of X and all the probabilities have to sum up to 1:

$$p(x) \in [0, 1] \text{ for each } x \in X$$
$$\sum_{x \in X} p(x) = 1$$

3.1.2. Probability models

A probability distribution over a set X is an instance of a *probability model*. The set of all probability distributions over a given set X is called the *unrestricted probability model* $\mathcal{M}(X)$.

$$\mathcal{M}(X) := \{p : X \to \mathbb{R} \mid p \text{ is a probability distribution}\}$$

A non-empty subset $\mathcal{M} \subset \mathcal{M}(X)$ is called *restricted probability model*. It can be used for integrating information we already know about our model, like dependencies between the probabilities.

As an example we could take a loaded die. As the die should still look like a normal die it is of course physically impossible for the probabilities of each side to get too low. So we could assume that each number has to occur with a probability of at least 0.1. This also results in a maximum probability of 0.5 on each side. So our model over the set $\mathcal{D} = \{1, \ldots, 6\}$ would be:

$$\mathcal{M} = \{p \in \mathcal{M}(\mathcal{D}) \mid \forall d \in \mathcal{D} : 0.1 \leq p(d) \leq 0.5\}$$

3.1.3. Conditional probability distribution

Conditional probability distributions are defined quite similar to probability distributions. Given two sets A and B, a probability distribution on B is assigned to each element

$a \in A$. For a given a and a given $b \in B$ this could be written as $(p(a))(b)$ but is mostly shortened to $p(b \mid a)$, which is read as "p of b given a". p has the following properties:

$$p : A \to \mathcal{M}(B)$$

$$p(b \mid a) \in [0, 1]$$

$$\forall a \in A : \sum_{b \in B} p(b \mid a) = 1$$

The unrestricted probability model for this conditional probability distribution is written as $\mathcal{M}(B \mid A)$.

3.2. Definition of probabilistic context-free grammars

A *probabilistic context-free grammars* (short: PCFG) is a 5-tuple consisting of the four elements of a CFG and a conditional probability distribution over the rules:

$$G := (\mathcal{V}, \mathcal{T}, S_0, \mathcal{R}, p)$$

p is a conditional probability distribution over $\mathcal{R}$. A rule $R \in \mathcal{R}$ is only applicable if the leftmost variable A corresponds to the syntactical category of R, thus $left(R) = \text{A}$.

Therefore we define:

$$p : \mathcal{V} \to \mathcal{M}(\mathcal{R})$$

$$p(R|\text{A}) := \begin{cases} 0, & \text{if left}(R) \neq \text{A} \\ p \in [0, 1], & \text{if left}(R) = \text{A} \end{cases}$$

As $p(R|\text{A})$ is only interesting for one A, namely left(R), we will write short for it:

$$p(R) := p(R|\text{A})$$

3.2.1. Probability of a derivation

As we have already seen in 2.3, parse trees are equivalent to derivations. Logically the probability of a derivation should be the same as the one of its parse tree. We have defined derivations in a way that each rule of a derivation has to be applicable. This means: Let $d = R_1 \dots R_n$ be a derivation. After the application of the first i rules $(0 < i < n)$, the leftmost variable is left(R_{i+1}).

So we can say:

$$p(d) = \prod_{i=1}^{n} p(R_i|\text{left}(R_i))$$

This definition fits to what we expect: if a rule with a low probability is applied, the probability of the derivation gets smaller than if one with a high probability is applied. Furthermore, the application of more rules (mostly yielding longer sentences) leads to a smaller probability.

As a sentence produced by a grammar often has different derivations, we also want to assign a probability to it. Let S_0 be the starting symbol, w be a sentence and p be a conditional probability distribution as described in 3.1.3.

$$p(w) := p(V \Rightarrow^* w) := \sum_{\substack{d \text{ with } \text{left}(d)=S_0, \\ \text{right}(d)=w}} p(d)$$

Again this makes sense: If we look at our example, "I saw her duck", one derivation would only cover one of the three meanings. As we want to consider all the meanings, we will have to sum up the probabilities.

3.2.2. An example

Recalling our example with the duck, we would now finally like to know which is the best derivation. For this, we take the rules which can be read off from the three "duck-trees" and assume that someone has given us probabilities[2] for them, which are written behind the sharp. Note that the probabilities of the rules for each syntactical category add up to 1.0. The rule "VBP → see" was added to make it clearer that "to saw" is less frequent than "to see".

S	→	NP VP	# 1.0	VP	→	VBD NP NN	# 0.4
NP	→	PRP	# 0.7	VP	→	VB	# 0.1
NP	→	PP$	# 0.3	VB	→	duck	# 1.0
PRP	→	I	# 0.8	VBP	→	see	# 0.9
PRP	→	her	# 0.2	VBP	→	saw	# 0.1
PP$	→	her	# 1.0	VBD	→	saw	# 1.0
VP	→	VBD NP VP	# 0.1	NN	→	duck	# 1.0
VP	→	VBP NP NN	# 0.4				

As all the probabilities of the rules in the trees are multiplied and some parts of the trees are equal, there is a constant part p_c, which we will calculate first.

$$p_c = p(\text{S} \rightarrow \text{NP VP}) \cdot p(\text{NP} \rightarrow \text{PRP}) \cdot p(\text{NP} \rightarrow \text{PP}) = 0.56$$

[2] how to calculate them is described in Chapter 4

The probabilities of the three trees are:

$$\begin{aligned}
p(\text{she ducks}) =& p_c \cdot p(\text{VP} \to \text{VBD NP VP}) \cdot p(\text{VBD} \to \text{saw}) \cdot p(\text{NP} \to \text{PRP}) \\
& \cdot p(\text{PRP} \to \text{her}) \cdot p(\text{VP} \to \text{VB}) \cdot p(\text{VB} \to \text{duck}) = \mathbf{7,84 \cdot 10^{-4}} \\
p(\text{tool}) =& p_c \cdot p(\text{VP} \to \text{VBP NP NN}) \cdot p(\text{VBP} \to \text{saw}) \cdot p(\text{NP} \to \text{PP\$}) \\
& \cdot p(\text{PP\$} \to \text{her}) \cdot p(\text{NN} \to \text{duck}) = \mathbf{6.7 \cdot 10^{-3}} \\
p(\text{see the bird}) =& p_c \cdot p(\text{VP} \to \text{VBD NP NN}) \cdot p(\text{VBD} \to \text{saw}) \cdot p(\text{NP} \to \text{PP\$}) \\
& \cdot p(\text{PP\$} \to \text{her}) \cdot p(\text{NN} \to \text{duck}) = \mathbf{6.7 \cdot 10^{-2}}
\end{aligned}$$

So we can decide that the most likely parse tree is the one, where the person sees her bird. We have found a way of disambiguating the sentence, but yet we cannot be sure whether we are right, as the result depends on the data which was used for annotating the rules with probabilities.

3.3. Corpora

For the annotation of the rules we cannot simply invent probabilities as it was done in the example before, but we need to calculate them on the basis of sample data. Such training data is called *corpus*. A corpus can be seen in three ways. It can be

1. a sequence of *data types*[3], which for example can be sets of words, phrases or sentences of a language or also rules of a context-free grammar.
2. a multiset of data types
3. a normalized multiset of data types

The first type could be the raw input for a training algorithm that counts the data types which results in the second type. A multiset-corpus c on a set X of data types is defined as follows:

$$c : X \to \mathbb{R}_{\geq 0}$$

The size of such a corpus is defined as

$$|c| := \sum_{x \in \mathcal{X}} c(x)$$

The corpus size is also called *quality* as a bigger corpus potentially delivers more information and having more statistical samples results in errors in the data to be less severe.

Dividing each value $c(x)$ by the corpus size, we get a normalized corpus. It corresponds to the relative frequency distribution of the corpus, which is a probability distribution as defined in 3.1.1.

[3]The set of data types which occur in a corpus is considered to be finite.

3.4. Parsing

Given a probabilistic context-free grammar, we do not only want to decide the word problem, i.e. the question whether a sentence can be derived or not, but also to evaluate the quality of a sentence or a sub-sentence and get its best parse tree. This is called stochastic parsing. For us the selection of the best parse tree is not important, as we want to calculate the weights or counts of the rules.

In the following two sections, the calculation of the inside- and outside weights of a sentence will be presented along with procedural algorithms for the computation. Those are taken from (Prescher 2002, 91-94, 139-144).

3.4.1. Inside parsing

The *inside* algorithm, based on the CYK algorithm (Kasami 1965; Younger 1967), calculates the probability of a sentence and of its sub-sentences starting with the probabilities of its words and then increasing the size of the subsets iteratively.

However the algorithm suffers a big problem: It operates using a grammar in *Chomsky normal form* (short: CNF) (Chomsky 1959). This means that the grammar only consists of the following two types of rules (with A, B, C being variables and a being a terminal):

$$A \to BC$$
$$A \to a$$

Epsilon rules, chain rules and identical rules are not allowed. It is easy to imagine that this does not reflect natural language too well and might yield analyzes which cannot easily be converted back to the original grammar. Moreover, though each grammar can be converted to the Chomsky normal form, this can result in an exponential gain of size.

The inside algorithm takes a PCFG $G = (\mathcal{V}, \mathcal{T}, S_0, \mathcal{R}, p)$ and a sentence $w = w_1 \dots w_n$ as input and outputs a *parse forest* consisting of all parse trees of each subset of w and their respective probabilities. The parse forests of a sentence $w_1 \dots w_n$ are denoted as:

$$\mathcal{F}_{\text{inner}}(s, t, \mathrm{A}) := \left\{ x \in T(G) \,\middle|\, \begin{matrix} \text{head}(x) = \mathrm{A} \\ \text{yield}(x) = w_s \dots w_t \end{matrix} \right\}$$

whereas $1 \leq s \leq t \leq n, A \in \mathcal{V}$. The inside forests are illustrated in Figure 3.1. The probability of an inside parse forest is named as follows:

$$p(\mathcal{F}_{\text{inner}}(s, t, \mathrm{A})) := \text{in}(s, t, \mathrm{A})$$

The values of $\text{in}(s, t, \mathrm{A})$ are called *inside probabilities* and can be calculated from the bottom up. They are defined as:

$$\text{in}(s, t, \mathrm{A}) := \begin{cases} p(\mathrm{A} \to w_s) & \text{if } s = t \\ \sum\limits_{\mathrm{A} \to \mathrm{BC}} \sum\limits_{i=s}^{t-1} p(\mathrm{A} \to \mathrm{BC}) \cdot \text{in}(s, i, \mathrm{B}) \cdot \text{in}(i+1, t, \mathrm{C}) & \text{else} \end{cases}$$

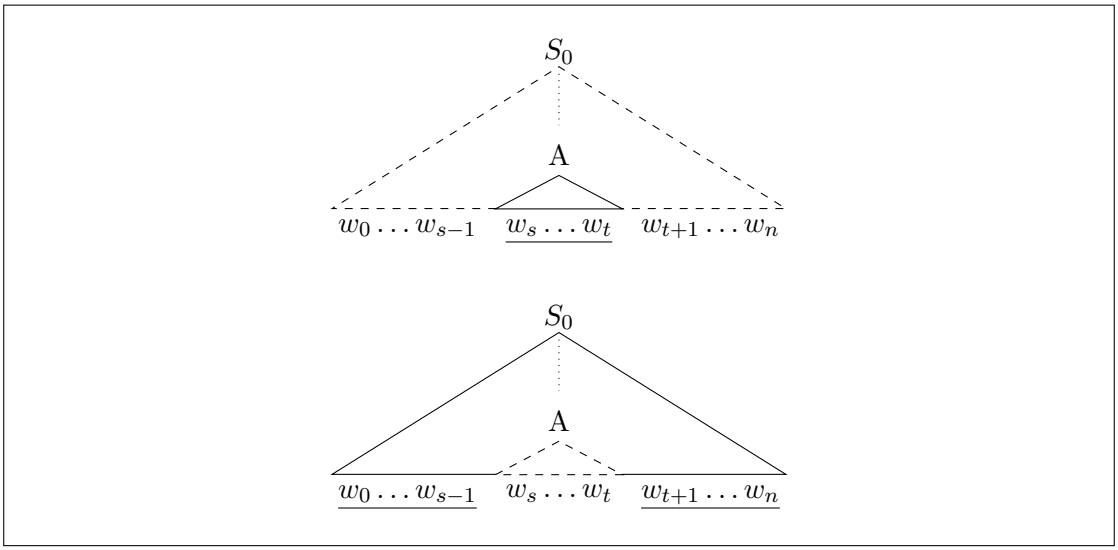


Figure 3.1.: Top: $\mathcal{F}_{\text{inner}}(s,t,\text{A})$: all possible trees with the root A, generating the subsentence $w_s \dots w_t$
Bottom: $\mathcal{F}_{\text{outer}}(s,t,\text{A})$: all trees starting from S_0, leaving out their subtrees starting with A, which yield $w_s \dots w_t$

This definition benefits of the property of grammars in CNF to only have two different types of rules.

Algorithm 3.1 is a slightly modified version of the one written by Prescher (2002, p.91). It can easily be seen that its complexity is $\mathcal{O}(n^3)$. The value m in line 7 determines, which part of the word is to be covered by the B-part of the rule and which part by the C-part.

3.4.2. Outside weights

The contrary to inside parse trees are *outside parse trees*. They are defined as follows:

$$\mathcal{F}_{\text{outer}}(s,t,\text{A}) := \left\{ x \in T(G) \middle| \begin{array}{l} \text{head}(x) = S_0 \\ \text{yield}(x) = w_1 \dots w_{s-1}\ \text{A}\ w_{t+1} \dots w_n \end{array} \right\}$$

with $1 \leq s \leq t \leq n$. The outside probabilities are named similar to the inside probabilities:

$$p(\mathcal{F}_{\text{outer}}(s,t,\text{A})) := \text{out}(s,t,\text{A})$$

Calculating the outside probabilities is not as easy as it was the case with inside probabilities. We need to know the inside probabilies first. Again, we have to differentiate between two cases:

Algorithm 3.1 The calculation of the inside weights

Input: A PCFG in Chomsky normal form and a sentence $w = w_1 \dots w_n$

Output: The inside probabilities $\text{in}[s, t, \text{A}]$ of all subsentences of the length $1 \leq t - s \leq n$ of all syntactic categories A.

1: $in[.,.,.] := 0$
2: **for all** $s := 1, \dots, n$ **do**
3: **for all** $\text{A} \to w_s$ **do**
4: $\text{in}[s, s, \text{A}] := p(\text{A} \to w_s)$
5: **for all** $l := 2, \dots, n$ **do** ▷ Lenght of the subsentence
6: **for all** $s := 1, \dots, n + 1 - l$ **do** ▷ Start of the subsentence
7: **for all** $m := 1, \dots, l - 1$ **do** ▷ Middle of the subsentence
8: **for all** $\text{A} \to \text{BC}$ **do**
9: $\text{in}[s, s + l, \text{A}] := \text{in}[s, s + l, \text{A}] +$
10: $p(\text{A} \to \text{BC}) \cdot \text{in}[s, s + m, \text{B}] \cdot \text{in}[s + m + 1, s + l, \text{C}]$

if $s = 1, t = n$:

$$\text{out}(s, t, \text{A}) := \begin{cases} 1 & \text{if A=S} \\ 0 & \text{else} \end{cases}$$

if $1 \leq s \leq t \leq n$ and $1 < s$ or $t < n$

$$\text{out}(s, t, \text{A}) := \sum_{\text{B,C} \in \mathcal{V}} \left(\sum_{r=1}^{s-1} \text{out}(r, t, \text{C}) \cdot p(\text{C} \to \text{B A}) \cdot \text{in}(r, s-1, \text{B}) + \sum_{r=t+1}^{n} \text{out}(s, r, \text{C}) \cdot p(\text{C} \to \text{A B}) \cdot \text{in}(t+1, r, \text{B}) \right)$$

The two sums inside the brackets stand for the two possible forms of the rule R which produces A: A can be the first or the second symbol generated by R. Now taking only the outside weight of the parent symbol of A would be wrong, as we keep the branch, which produces the sibling of A out. This is illustrated in Figure 3.2.

Algorithm 3.2 calculates the outside weights with a complexity of $\mathcal{O}(n^3)$ (Prescher 2002).

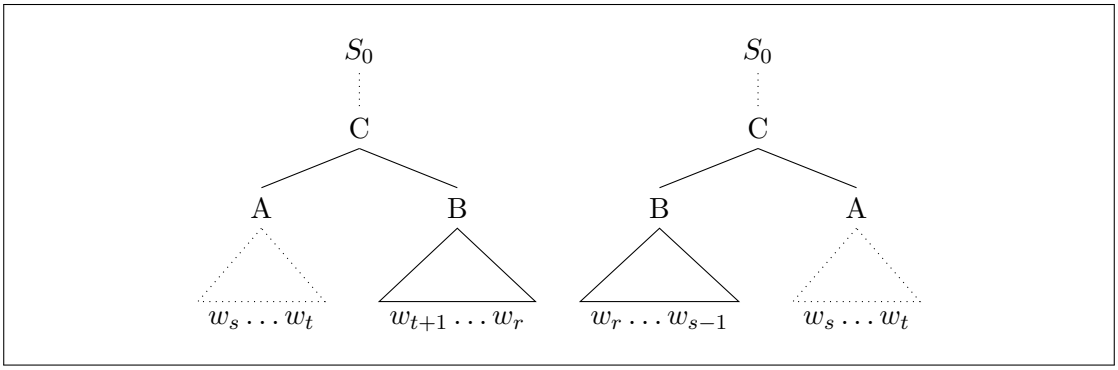


Figure 3.2.: The two possibilities how the outside tree could look.

Algorithm 3.2 The calculation of the outside weights

Input: A PCFG in Chomsky normal form and a sentence $w = w_1 \dots w_n$
Output: The outside probabilities $\text{out}[s, t, \text{A}]$ with $1 \leq s \leq t \leq A$.

1: $\text{out}[., ., .] := 0$
2: $\text{out}[1, n, S_0] = 1$
3: **for all** $l := n - 1, \dots, 1$ **do**
4: **for all** $s := 1, \dots, n - l$ **do**
5: **for all** $m := 0, \dots, l - 1$ **do**
6: **for all** $\text{A} \to \text{BC}$ **do**
7: $\text{out}[s, s + m, \text{B}] := \text{out}[s, s + m, \text{B}] + p(\text{A} \to \text{BC}) \cdot \text{out}[s, s + l, \text{A}] \cdot \text{in}[s + m + 1, s + l, \text{C}]$
8: $\text{out}[s + m + 1, s + l, \text{C}] := \text{out}[s + m + 1, s + l, \text{C}] + p(\text{A} \to \text{BC}) \cdot \text{out}[s, s + l, \text{A}] \cdot \text{in}[s, s + m, \text{B}]$

4 Training with the EM algorithm

4.1. Maximum Likelihood Estimation

For measuring the quality of a probability distribution over a corpus, we can use the *corpus likelihood*, which should be as big as possible. For a corpus c, it is defined as

$$L_p(c) = \prod_{x \in \mathcal{X}} p(x)^{c(x)}$$

Maximum likelihood estimation aims at finding the probability distribution $mle(f, \mathcal{M}) \in \mathcal{M} \subset \mathcal{M}(\mathcal{X})$ which maximizes the corpus likelihood.

$$\text{mle}(c, \mathcal{M}) = \text{argmax}_{p \in \mathcal{M}} L_p(c)$$

Dividing the frequencies by the size of a corpus f results in a probability distribution as defined in 3.1.1. As it is the direct representation of empirical data it is called *empirical probability distribution*:

$$p : \mathcal{X} \to [0, 1] \text{ where } p(x) = \frac{f(x)}{|f|}$$

Now interestingly $\text{mle}(c, \mathcal{M}) = p$. This means, that the empirical probability distribution represents the corpus best. This is proven by Prescher (2005, Chapter 2.4).

This can also be extended to conditional probability models. Given a set $\mathcal{X}$, a set $\mathcal{Y}$, a corpus $c' : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}_{\geq 0}$ and a probability model $\mathcal{M} \subset \mathcal{M}(Y|X)$

The maximum likelihood estimate now is:

$$\text{mle}(c', \mathcal{M}) := \text{argmax}_{p \in \mathcal{M}} L_p(c')$$

where

$$L_p(c') := \prod_{(x,y) \in \mathcal{X} \times \mathcal{Y}} p(y \mid x)^{c(x,y)}$$

The empirical probability distribution on c' is calculated as follows:

$$p' : \mathcal{X} \to \mathcal{M}(\mathcal{Y}) \text{ where } p'(y \mid x) = \frac{c'(x, y)}{\sum_{y' \in \mathcal{Y}} c'(x, y')}$$

p' again is the maximum likelihood estimate on the corpus:

$$\text{mle}(c', \mathcal{M}) := p'$$

4.2. The EM algorithm

The *expectation-maximization* algorithm (shortened as EM algorithm) (Dempster, Laird, and Rubin 1977; Prescher 2005) aims at finding the "best fitting" probability distribution of a probability model. This means it tries to maximize the corpus likelihood defined in the formula 4.1. Basically this is done in two (in the first instance) infinitely alternating steps. Thereby the *expectation* step calculates the complete data corpus and the *maximization* step computes the maximum likelihood estimate of the model on the complete data corpus. The algorithm has to be terminated "manually" after a fixed amount of steps or, better, after the results do not significally change anymore. As we will see later, the algorithm does not find the global but only a local maximum, which depends on what parameters are selected as an input.

In the following sections we will first define the general form of the algorithm and then instantiate it.

4.2.1. Prerequisites

The EM algorithm takes several inputs. First of all, it needs a corpus $h : X \to \mathbb{R}_{\geq 0}$. The set X depends on the instance and is called the set of *incomplete data types*.

We also need a set Y of *complete data types*, to which the incomplete data types are related by a function called yield[1].

$$\text{yield} : Y \to X$$

yield has to be surjective, which means that we can also define a function yield^{-1}:

$$\text{yield}^{-1}(x) := \{y \in Y \mid \text{yield}(y) = x\}$$

Next, we will need a probability model $\mathcal{M} \subseteq \mathcal{M}(Y)$. It is a restricted probability model as we usually already know something about the structures we want to train using the

[1] Defining an instance the selection of this name will become clearer: in many cases it describes the yield, i.e. a sentence, of a tree.

algorithm. Therefore we need three different sets, A, B and C. A and B depend on which instance of the algorithm we have and $C \subseteq A \times B$.

So how do these sets relate to the data types? For this, we need a function $\pi : Y \to (A \times B)^*$. In the instances of the algorithm, π will strip down complete data types to chunks in C. The probabilities of those chunks later are the wanted result of the algorithm. The conditional probability distribution over A and B is a $p \in \mathcal{M}(A \mid B)$. We define:

$$p^\pi : Y \to \mathbb{R}_{\geq 0}$$
$$p^\pi(y) := \prod_{i=1}^{n} p(a_i \mid b_i)$$
$$\text{where } \pi(y) = (a_1 \mid b_1) \dots (a_n \mid b_n)$$

The last equation means that π is applied to y and the product function multiplies the probabilities of each element of the resulting sequence.

Now we first define the restricted model $\mathcal{M}_C(A \mid B) \subseteq \mathcal{M}(A \mid B)$:

$$\mathcal{M}_C(A \mid B) := \{p \in \mathcal{M}(A \mid B) \mid \forall (a, b) \in (A \times B) \setminus C : p(a \mid b) = 0\}$$

For this model we can now define $\mathcal{M}_C(A \mid B)^\pi \subseteq \mathcal{M}(Y)$:

$$\mathcal{M}_C(A \mid B)^\pi := \{p^\pi \mid p \in \mathcal{M}\}$$

The last prerequisite for the algorithm is an initial probability distribution $p_0 \in \mathcal{M}$, which, as described, in fact is a distribution in $\mathcal{M}_C(A \mid B)$, just mapped by π to a distribution in $\mathcal{M}(Y)$. It has to be kept in mind, that each of the following distributions p_i will be such a distribution.

4.2.2. The algorithm

Now the algorithm computes iteratively probability distributions p_i in two alternating steps: The *expectation step* (E-step) and the *maximization step* (M-step). The E-step computes the complete-data corpus h_q and the M-step computes its maximum likelihood estimate, which is the new probability distribution p_{i+1}. The output of the algorithm is the series of p_i.

The probability distribution p_i of the current step will be called q. Now the complete-data corpus $h_q : Y \to \mathbb{R}_{\geq 0}$ is calculated as follows:

$$h_q(y) := h(\text{yield}(y)) \cdot \frac{q(y)}{\sum_{y' \text{ with } \text{yield}(y)=\text{yield}(y')} q(y')}$$

Now, for calculating the probability distribution p_{i+1} we first need to extract the corpus $h' : A \times B \to \mathbb{R}_{\geq 0}$:

$$h'(a, b) := \sum_{y \in Y} h_q(y) \cdot \#_{(a,b)}(\pi(y))$$

Here $\#_{(a,b)}$ returns the count of the pairs (a, b) in $\pi(y)$. Now, we need to calculate the maximum likelihood estimate on h_q, which again is induced by a maximum likelihood estimate on $h'(a, b)$.

$$p_i := \text{argmax}_{p \in \mathcal{M}} L_p(h_q)$$

Algorithm 4.1 The EM algorithm

Input: corpus $h : X \to \mathbb{R}$ on incomplete data types
set of complete data types Y
probability model $\mathcal{M} = \mathcal{M}_C(A \mid B)^\pi \subseteq \mathcal{M}(Y)$
initial distribution $p_0 \in \mathcal{M}$

Output: sequence $p_0, p_1, \ldots$ of $p_i \in \mathcal{M}$

1: **for all** $i = 1, 2, \ldots$ **do**
2: $q := p_{i-1}$
3: E-step: compute complete-data corpus $h_q : Y \to \mathbb{R}$
4:

$$h_q(y) = h(\text{yield}(y)) \cdot \frac{q(y)}{\sum_{y' \in \text{yield}^{-1}(\text{yield}(y))} q(y')}$$

5: M-Step: compute maximum likelihood estimate of $\mathcal{M}$ on h_q
6:

$$p_i = \text{argmax}_{p \in \mathcal{M}} L_p(h_q)$$

7: output p_i

4.2.3. Training of PCFG on sentence corpora as an instance of the EM algorithm

Prescher (2005) describes a form of training PCFG on a sentence corpus which is explained here. We will first instantiate the inputs (the grammar which is being trained is $G = (\mathcal{V}, \mathcal{T}, S_0, \mathcal{R})$):

- $X = \mathcal{L}_G$
- $Y = D_G$
- $C = \mathcal{R}$ with $A = (\mathcal{V} \cup \mathcal{T})^*$ (right sides of the rules) and $B = \mathcal{V}$ (left sides)
- yield(y) returns the sentence $x \in X$ which is the result of the conjunction of all leaf nodes of y, read from left
- π only formats the rules of a derivation $d = R_0 \ldots R_n$:

$$\pi(d) = (\text{right}(R_0), \text{left}(R_0)), \ldots, (\text{right}(R_n), \text{left}(R_n))$$

- the probability distributions $p^\pi \in \mathcal{M}_C(A|B)^\pi$ are therefore defined as:

$$p^\pi(R_1 \dots R_n) = \prod_{i=1}^{n} p(\text{right}(R_i) \mid \text{left}(R_i))$$

Algorithm 4.2 Training PCFG using the EM algorithm

Input: G, p_0 as already defined
Output: sequence $p_i \in \mathcal{M}_C(A|B)^\pi$

1: **for all** $i = 1, 2, \dots$ **do**
2: $q := p_{i-1}$
3: E-step: compute complete-data corpus $h_q : D_G \to \mathbb{R}_{\geq 0}$
4:

$$h_q(d) = h(\text{right}(d)) \cdot \frac{q(d)}{\sum_{d' \text{ with right}(d')=\text{right}(d)} q(d')}$$

5: M-Step: compute maximum likelihood estimate of $\mathcal{M}$ on h_q
6:

$$h'(R) = \sum_{d \in D_G} h_q(d) \cdot \#_{(\text{right}(R),\text{left}(R))}(\pi(d))$$

7:

$$p_i(R) = \frac{h'(R)}{\sum_{R' \text{ with left}(R')=\text{left}(R)} h'(R')}$$

8: output p_i

Taking a closer look at the algorithm, we can figure out a major problem: The whole process is based on all derivations of the grammar, which is a potentially infinite amount. We would like to reduce this list to just the derivations we really need, i.e. the derivations of our parse trees. There are two possibilies for that:

1. manually assigning only one parse tree to each sentence (meaning that a tree corpus is used)
2. calculating all the parse trees for each sentence

In the following sections those two methods will be handled more explicitely.

4.3. Supervised training

It seems to be a good idea to choose parse trees as training data, which allows us to skip the difficult step of parsing the sentences. As the sentences are labeled by their parse trees, training a grammar on a tree corpus is a form of *supervised training*. As we will see, this makes the training process very simple. However preparing a tree corpus is very costly, as the labeling has to be done manually, which is fault-prone and time consuming.

The procedural algorithm 4.3 for supervised training only consists of two steps: Counting all the rules (the counts are named $C[R_i]$ for rules R_i and $C[A_i]$ for syntactical categories A_i) and then calculating the respective conditional relative frequencies.

Algorithm 4.3 Supervised training of a PCFG

Input: A CFG $G = (\mathcal{V}, \mathcal{T}, S_0, \mathcal{R})$
A corpus of trees/derivations $h : D_G \to \mathbb{R}_{\geq 0}$
Output: A PCFG G'

```
for all R in R do
    C[R] := 0
    C[A] := 0
for all d in h do
    for all R in readoff(d) do
        C[R] := C[R] + h(R)
        C[left(R)] := C[left(R)] + h(d)
for all R in R do
    p(R) := C[R] / C[left(R)]
return G' = (V, T, S_0, R, p)
```

In fact, the algorithm is very similar to the EM algorithm: Already having a tree corpus h_q over D_G, we don't need a corpus h over $\mathcal{L}(G)$ anymore, as it is only used for calculating h_q. So the algorithm consists only of an M step, which is kept unchanged. The formula

$$h'(R) = \sum_{d \in D_G} h_q(d) \cdot \#_{(\text{right}(R),\text{left}(R))}(\pi(d))$$

calculates the counts of the rules and

$$p_i(R) = \frac{h'(R)}{\sum_{R' \text{ with } \text{left}(R')=\text{left}(R)} h'(R')}$$

calculates the according probabilities, just as defined in the procedural form. The algorithm also can be terminated after one iteration as h_q is fixed and especially independent of the new probability distribution p_1.

4.4. Unsupervised training with the inside-outside-algorithm

Unsupervised training, in contrary to supervised training, uses an unlabeled corpus. Thus it works a lot like Algorithm 4.2, except that it uses inside-outside-parsing for calculating the probabilities. The algorithm therefore is called *inside-outside algorithm*. It is based on the forward-backward algorithm by Baum (1972) and Baker (1979) respectively and was was generalized to allow training corpora with more than one sentence by Lari and Young (1990).

4.4.1. Counts

An important part of the algorithm will be the accumulation of the results of the inside-outside-parsing for each sentence. The values of C_w are the rule counts for each sentence and the counts of their syntactic categories.

$$
\begin{aligned}
p(w) &:= \mathrm{in}(1, n, S_0) \\
C_w(\mathrm{A} \to \mathrm{a}) &:= \frac{1}{p(w)} \sum_{\substack{1 \le t \le n, \\ w_t = \mathrm{a}}} \mathrm{in}(t, t, \mathrm{A}) \cdot \mathrm{out}(t, t, \mathrm{A}) \\
C_w(\mathrm{A}) &:= \frac{1}{p(w)} \sum_{s=1}^{n} \sum_{t=s}^{n} \mathrm{in}(s, t, \mathrm{A}) \cdot \mathrm{out}(s, t, \mathrm{A}) \\
C_w(\mathrm{A} \to \mathrm{BC}) &:= \frac{1}{p(w)} \sum_{s=1}^{n-1} \sum_{t=s+1}^{n} \sum_{r=s}^{t-1} p(\mathrm{A} \to \mathrm{BC}) \\
&\quad \cdot \mathrm{in}(s, r, \mathrm{B}) \cdot \mathrm{in}(r+1, t, \mathrm{C}) \cdot \mathrm{out}(s, t, \mathrm{A})
\end{aligned}
$$

For each rule R the new probability is defined as follows using the re-estimation rules in the form described by Lari and Young (1990):

$$
p(R) := \frac{\sum_{w=y_1}^{y_N} C_w(R)}{\sum_{w=y_1}^{y_N} C_w(\mathrm{left}(R))}
$$

Prescher (2002, 228-234) proved that $p(R)$ in fact is a probability distribution. Therefore for all A in $\mathcal{V}$:

$$
\begin{aligned}
1 &= \sum_{R \text{ with } \mathrm{left}(R)=\mathrm{A}} p(R) \\
&= \frac{\sum_{R \text{ with } \mathrm{left}(R)=\mathrm{A}} \sum_{w=y_1}^{y_N} C_w(R)}{\sum_{w=y_1}^{y_N} C_w(A)} \\
&= \frac{\sum_{R \text{ with } \mathrm{left}(R)=\mathrm{A}} \sum_{w=y_1}^{y_N} C_w(R)}{\sum_{R \text{ with } \mathrm{left}(R)=\mathrm{A}} \sum_{w=y_1}^{y_N} C_w(R)} = 1
\end{aligned}
$$

Thus:

$$
\sum_{w=y_1}^{y_N} C_w(A) = \sum_{R \text{ with } \mathrm{left}(R)=\mathrm{A}} \sum_{w=y_1}^{y_N} C_w(R)
$$

This means that for getting the counts for the grammatical categories, we can simply sum up the rule counts which makes the algorithm more efficient than the one proposed by Prescher or Lari and Young.

4.4.2. The algorithm

Algorithm 4.4 is, as already mentioned, an instance of the EM-algorithm. This is proven by Prescher (2001).

Again this algorithm is written in procedural form, which makes it optically differ a lot from Algorithm 4.2. The main difference is that in the E-Step we do not calculate the complete data corpus h_q but the counts of the rules in it, which in fact is the rule corpus h'. We also count the number of rules for each syntactic category. This has three advantages:

1. We do not need to consider every possible derivation of D_G, which would, using a realistic grammar, be at least very big, if not infinite.
2. Computing h' we do not have to look at each derivation again for each single rule.
3. Already having the counts for each syntactic category, we do not need to calculate them again in the M-Step.

In the M-Step we simply calculate the relative frequencies of the rules in C under the condition of their respective syntactic categories.

Algorithm 4.4 Training of a PCFG with the inside-outside algorithm

Input: A CFG $G = (\mathcal{V}, \mathcal{T}, S_0, \mathcal{R})$
sentence corpus h with the size $|h|$ consisting of sentences $h_1, \ldots, h_n$
initial probability distribution p_0
Output: series of probability distributions $p_1 \ldots p_n$

for all $i := 1, 2, \ldots$ **do**
  for all $R \in \mathcal{R}$ **do**
    $C[R] := 0$
  $q := p_i$
  E-Step:
  for all $j := 1, \ldots, |h|$ **do**
    $w := c_j$
    compute in(.,.,.), out(.,.,.)
    for all $R \in \mathcal{R}$ **do**
      compute $C_w(R)$
      $C[R] = C[R] + C_w(R)$
      $C[\text{left}(R)] = C[\text{left}(R)] + C_w(R)$
  M-Step:
  for all $R \in \mathcal{R}$ **do**
    $p_i(R) := \frac{C[R]}{C[\text{left}(R)]}$
  print p_i

4.4.3. Termination of the algorithm

As the inside-outside-algorithm does not have a termination criteria itself, this has to be considered in the implementation. There are two obvious possibilities for that:

1. Terminating after the results do not significantly change. Though this sounds reasonable, it comes with some problems: Not only has each probability distribution (or the corpus likelihood) to be compared with its predecessor, there also is the chance that the probabilities just are lying in a plateau and will change significantly after some steps.[2] It is also rather unpretendable, how long the algorithm would run.
2. Terminating after a fixed amount of steps. This amount has to be determined experimentally, but it keeps us from checking a complicated termination condition in each step.

4.4.4. Convergence characteristics

We hope for the algorithm to improve the resulting probability distribution with each iteration, so that we approach a local maximum in the corpus likelihood. This means that we want:

$$L_{p_{i-1}}(h) \leq L_{p_i}(h)$$

According to Prescher (2002) this is fulfilled, meaning that the algorithm improves the corpus likelihood.

4.4.5. Usage of the algorithm in practice

Using an implementation of the algorithm, like the one described in Chapter 5, we could simple use our whole corpus as training data and without further thinking let the algorithm run as long as possible. However as Prescher describes in his dissertation (Prescher 2002, 148-154), this is not the best approach.

First of all it is necessary to split the corpus into training data and test data. Often the division is 90% for training and 10% for evaluation. Prescher even goes further and splits off a development test corpus and a test corpus for a final evaluation. The two test corpora have to be manually annotated in order to be used.

Afterwards the training process is run multiple times, whereas three free parameters have to be varied: The size of the corpus, the number of iterations (by using the probability distribution generated in the intermediate steps of the EM algorithm) and the initial probability distribution. Lari and Young (1990) in contrary simply use the last probability distribution and do not stop the algorithm at a fixed number of iterations but after the difference between two consecutive distributions gets below a bound. They also only use a fixed initial grammar.

As the algorithm only converges to a local maximum, it is very likely that varying said parameters (especially the initial distribution) could lead to a better result. But also the

[2] In experiments of Lari and Young this actually is the case. (Lari and Young 1990, 47, Figure 5)

other parameters are important: Let us say, we were lucky and the algorithm even found the global maximum for one corpus. This does not necessarily have to be the best value for a different corpus. If this result has a rule with a probability close to zero but an entry of another corpus needs said rule, the (bigger) probability some steps before could result in a bigger likelihood for this other corpus.

So in the end of the training process we get a lot of different probability distributions. Those have to be evaluated using the development test data, whereas the best distribution is selected and again evaluated using the final test data.

For the evaluation itself there exist several approaches. A simple approach is just to check whether two parse trees are completely identical. As this alone is a very strong condition another method there also exist other possibilies for evaluation (Prescher 2002, 74f.). For example, if in the reference tree somewhere a variable V yields the words w_3 to w_6, in the parse which is to be evaluated, there are points given for whether the same sub-sentence is also produced by a single variable and also if the variable is the same. One could also give points for the correct tagging of the words and thereby to evaluate the severity of errors: still mistakenly tagging a noun as a verb is more severe than mixing up plural and singular (like with “sheep”, where singular and plural are equal).

5 Implementation for Vanda

Vanda[1] is a statistical machine translation developed at the Chair Foundations of Programming at the Dresden University of Technology. *Vanda studio* is an integrated development environment (IDE) which joins different applications for machine translation to allow doing small-scale experiments with them. The application can come as binary modules written in any language, but more importantly can imported from the accompanying haskell library *Vanda Toolkit*, for which the following implementation of the two training methods is written.

5.1. Data structures

The **PCFG** data structure consists of three parts storing the rules of the underlying context-free grammar, the weights or probabilities of the rules and a mapping of variables to integer values.

For fast lookups the latter is a Data.Vector. Its first entry is treated as the starting symbol of the grammar. The rules are stored in a vector, which contains a vector for each syntactic category. Those vectors again are ordered just as the mapping vector. Parallely to the rules, their respective probabilities are stored in a vector of exactly the same form.

The rules are **Hyperedge**s[2], which are implemented in **Vanda.Hypergraph**. Each of the **Hyperedge**s has a variable "to", which represents the left hand side of the rule. The inside-outside algorithm expects the grammar to be in Chomsky normal form. Therefore the edges can be of the type **Nullary**, with a terminal as label, or of the type **Binary**, without label and the two variables on the right hand side of the rule are stored in from1 and from2. The values of "ident" signify, at which position of the vectors a rule is located, so that the weights of a rule can be easily found. The two other egde types, **Unary** and **Hyperedge** are used for grammars which are not in CNF and which can be trained with the treebank algorithm.

An example for the data structure **PCFG** can be found in Figure 5.1.

[1] http://www.inf.tu-dresden.de/index.php?node_id=2550&ln=en

[2] What a hypergraph is, does not matter here, as only the edges are used, because they come with some practical access functions.

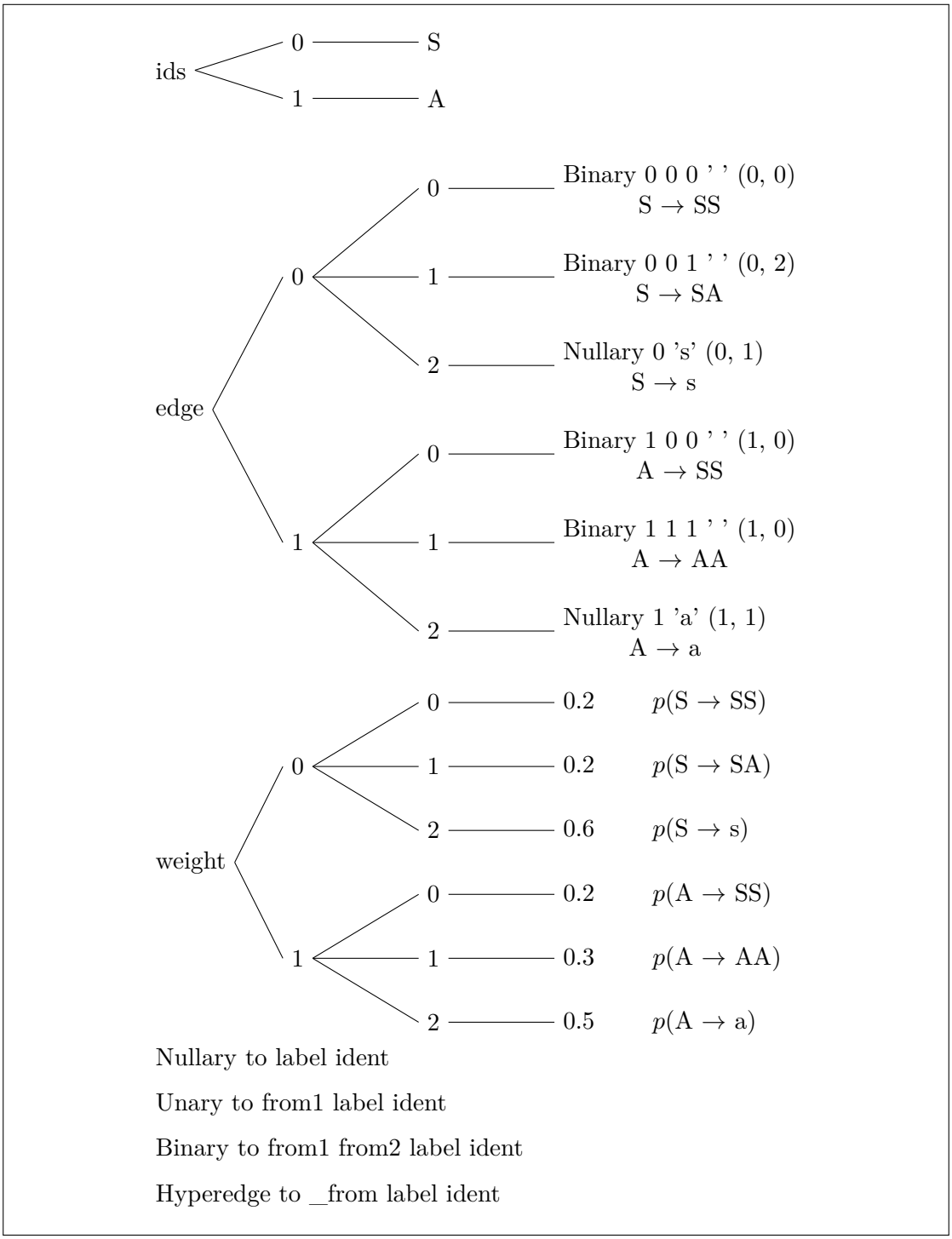


Figure 5.1.: The data structure for the example PCFG (in CNF, using the same CFG as the grammar used for testing in Section 5.5) in Vanda including the data constructors for the rules ("Hyperedges").

5.2. Maximum likelihood estimate

The function for the calculation of the maximum likelihood estimate of a given corpus is called **normalize**. It is very simple as the division of the rules into their syntactic categories makes counting and dividing them by the sum very easy. The calculation works for grammars no matter whether they are in Chomsky normal form or not because **normalize** only takes a weights-vector as input and returns a new one.

5.3. Supervised training

Supervised training besically is just a call of the function **normalize** with just the vector of weights as input. As it contains the probabilities of the rules ordered by their syntactical category, no knowledge of the underlying grammar is needed.

For some easier handling there also exists a set of function for parsing a tree bank from a text file. First of all there are two parsers for the conversion of a textfile (with one tree per line) to a list of trees. **parseTree1** parses trees in the format described in this work and **parseTree2** parses trees which are notated like this:

(S (NP (DT This)) (VP (VBZ is) (NP (DT another) (NN notation))) (. .))

The same tree in the notation used in this work would be:

S(NP(DT(This)) VP(VBZ(is) NP(DT(another) NN(notation))) .(.))

ruleMap takes such a list of trees and splits it to a hashmap which contains triples of the following form:

$$(\text{left}(R), [\text{var}], [\text{Either var term}])$$

The second value is the sequence of all variables of right(R) and the third one represents the whole right(R) using the Either data type, where the items of the type Right are variables and the ones of the type Left are terminals. The values of the map are the counts of the rules.

Finally, **ruleMapToPCFG** takes such a hashmap and produces a grammar out of it.

5.4. The inside-outside algorithm

5.4.1. Inside and outside weights

The central part of the inside-outside algorithm is the calculation of the inside and outside weights. The respective functions are called **inside** and **outside**.

inside takes a sentence (stored in a vector of words) and a **PCFG** as input. It returns a Data.Hashmap, which looks very similar to the formal definition of the inside function: The keys are triples of integers (s, t, a) where s and t are the same as in the formal definition (minus one) and a is the id of a variable taken from **ids** of the grammar.

outside additionally takes the result of an application of **inside** and returns a map similar to the one which is returned by **inside**.

The calculation of the weights is rather similar to the procedural algorithms 3.1 and 3.2. The two big differences are that first for easier reading in the pseudocode the numbering of the words starts with 1 and in the implementation with 0 and second the loops and adding up of the inside weights are summarized in list comprehensions.

5.4.2. Calculation of the counts

The calculation of the counts $C(R)$, or rather the probabilities induced by the counts, is done by the functions **count** and **counts**.

count calculates $C_w(R)$ for a sentence w. It takes a sentence (i.e. a vector of words) and a grammar as input and outputs $C_w(R)$ in the form of a weight vector. **count** calls **inside** and **outside** and calculates $C_w(R)$ as defined.

counts takes a corpus (i.e. a vector of senctences) and a grammar as input and outputs the new grammar with the normalized new weights. It calls **count** on each sentence and adds up all the values of the vectors and then calls **normalize**.

The function **counts** therefore in fact calculates a complete EM-step, where the weight vector of the input grammar is p_{i-1}.

5.5. Some tests

What makes PCFG interesting is their ability to disambiguate. For a test of this, very small data is used. The rules of our grammar look like this:

(1)	S	→ S S	(2)	S → S A	(3) S → s
(4)	A	→ S S	(5)	A → A A	(6) A → a

First of all, we take a closer look at the rules in order to get an idea how well they might be rated by the inside-outside algorithm. Rule (2) is very important, as it is the only one which leads us from S to A. We can also see, that we have to generate at least one s, which means that rule (3) also is important. If we have at least one a in our corpus, rule (6) will get a probability greater than 0. What about rule (5)? If we only have words with longer sequences of s than of a, it could be replaced using rule (2). If there are longer sequences of a, it will get more important.

In both cases one of the rules (1) or (4) should get a very low probability and the other one a quite high probability, so that we can sort the "bad" one out. But if we have the word "ss" in our corpus rule (1) should definitely get a higher probability.

What are the test results? First we will try the following corpus and use evenly distributed initial probabilitiies:

- s s s s
- s s s s s s s s

We get the following probabilities:

(1) 0.454	(2) 8.510e-7	(3) 0.545
(4) 1.0	(5) 2.334e-34	(6) 0.0

Rule (6) has a probability of 0, as no a occurs in the corpus. The really important rules, (1) and (3) get very similar probabilities, whereas the one for (3) is slightly higher. This makes the grammar consistent (Stüber 2012).

Rule (2) has a probability close to zero and rule (4) has a probability of 1, which means that in the very unlikely case of generating an A, it will surely be replace by two S again.

Another corpus could be:

- s a s a s a s a s a
- s s s a a a a a a a
- s s s s s s s s s a

Now we get the following result:

(1) 0.350	(2) 0.224	(3) 0.426
(4) 2.245e-11	(5) 0.237	(6) 0.763

It seems to be relatively unlikely to switch the syntactical category: rule (2) has no high probability and the probability of rule (4) is close to zero. So it clearly looks like rule (1) is the "better" one for generating s.

What happens if we take a different initial probability distribution? We will use this distribution, which rates rule (1) worse and rule (4) better from the beginning:

(1) 0.2	(2) 0.5	(3) 0.3
(4) 0.5	(5) 0.2	(6) 0.3

This gives us another result as before, which shows us, that the initial probability distribution is an important parameter for the training.

(1) 0.0	(2) 0.537	(3) 0.463
(4) 0.345	(5) 1.510e-2	(6) 0.640

The initially better rating of rule (4) and the worse rating of rule (1) caused the probabilities of the rules to change significantly. And what happens, if we additionally insert the sentence “s s” into our corpus? As expected, regardless of how bad the rule (1) is rated initially, it always gets a similar probability as with the corpus without “s s” using the even distribution.

6 What comes next

In the next sections I will explain some of my thoughts about improving the training algorithms which could be a topic for further research. Those are only unproven ideas as finding proves for them would have exceeded the scope of this thesis. Some approaches found in other papers are also included.

6.1. Resolving the drawbacks of the Chomsky normal form

An important topic of further research is how to avoid or to deal with the necessity of using a grammar in Chomsky normal form.

6.1.1. Transforming parses back to the original grammar

An option could be to find a way to transform a parse tree back to one of the original grammar. The transformation to Chomsky normal form consists of five steps:

1. the treatment of the exception $S_0 \to \varepsilon$
2. the introduction of rules $V_a \to a$ for each terminal a and replacement of all terminals in rules R where right(R) contains at least one variable
3. the introduction of rules $V_{XY} \to XY$ for each pair of variables which occurs in rules R where right(R) has more than two variables and the replacement of the pairs
4. the removal of ε-rules
5. the removal of chain rules

If we do not allow ε-rules in the initial grammar, we can skip steps 1 and 4 and do not need to transform them back. This actually is no big problem, as those rules do not really represent natural language. Take the two rules A $\to$ BEC and E $\to \varepsilon$ as an example. We can also keep the first one and replace the second one by a modification of the first one where the E-symbol is missing: A $\to$ BC.

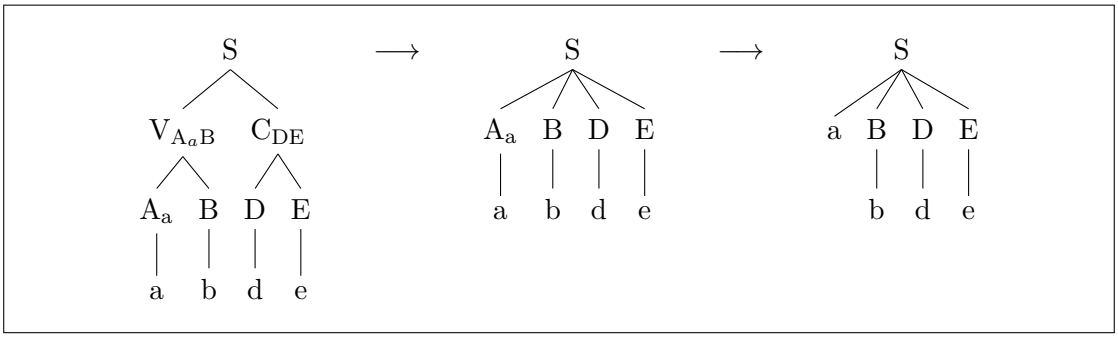


Figure 6.1.: The flattening of rules in the transformation from a parse tree in CNF back to the initial grammar

The rules introduced by steps 2 and 3 can also be flattened again, as the variables are labeled by their child symbols. This is illustrated in Figure 6.1.

The bigger problem is the elimination of chain rules in step 5. As the rules are removed, for transfering the parse tree back to the initial grammar we need to introduce rules again. Also the elimination itself could maybe result in a shift of the "meaning" of the original grammar, making it possible to train the probabilities wrong. Consider for example the following rules:

$$(1): \mathrm{A} \to \mathrm{B}$$
$$(2): \mathrm{A} \to \mathrm{AA}$$
$$(3): \mathrm{B} \to \mathrm{AA}$$

During the transformation, rule (1) will get deleted and a new rule will be introduced. This rule however is identical with rule (2) and cannot be distinguished using the definitions given in this thesis. Therefore we cannot decide whether the original chain rule (1) followed by rule (3) or only rule (2) has to be used when the parse tree is transformed back.

So technically, if we have a grammar without chain rules, we can transform it to Chomsky normal form and back. But it is yet to be proven, that there are no significant changes in the probabilities doing so.

6.1.2. Allowing a more general grammar using a different parsing algorithm

The inside and outside weights could be redefined to also allow longer rules, rules with mixed terminals and variables and chain rules. An approach to do this is the CYK+ algorithm by Chappelier and Rajman (1998). It allows all rules of CFG except mixed rules. This restriction is called nplCFG (non-partially lexicalized CFG) by the authors.

nplCFG represent the grammar behind the Penn Treebank well as each word is part-of-speech-tagged, meaning that mixed rules do not occur. The algorithm can even deal

with unknown words. They are relatively likely to occur in a corpus and would otherwise cause the sentence with the word to be "lost".

However the authors only mention the calculation of the inside weights. So using their calculation, the outside weights have to be redefined, too.

6.2. A mixed approach between supervised and unsupervised training

Zhou and Lua (1998) have found an interesting approach for training probabilistic context-free grammars. What if we already have a tree corpus, which is however too small to produce a well trained grammar? They use a relatively simple idea for that: first they calculate the initial probabilities using the tree corpus and use those training results as input for the unsupervised training, where in each step again the probabilities of the tree corpus are considered.

This works as some sort of a guideline for the unsupervised training. Based on experiments the writers of the paper conclude that this approach leads to a significant improvement of the training results.

References

Baker, J. K. (1979). Trainable grammars for speech recognition. In D. H. Klatt and J. J. Wolf (Eds.), *Speech Communication Papers for the 97th Meeting of the Acoustical Society of America*, pp. 547–550.

Baum, L. E. (1972). An inequality and associated maximization technique in statistical estimation for probabilistic functions of markov processes. *Inequalities 3*, 1–8.

Booth, T. and R. Thompson (1973). Applying probability measures to abstract languages. *IEEE Transactions on Computers C-22*(5), 442–450.

Booth, T. L. (1969). Probabilistic representation of formal languages. In *Proceedings of the 10th Annual Symposium on Switching and Automata Theory (swat 1969)*, SWAT '69, Washington, DC, USA, pp. 74–81. IEEE Computer Society.

Chappelier, J.-C. and M. Rajman (1998). A practical bottom-up algorithm for on-line parsing with stochastic context-free grammars.

Chomsky, N. (1959). On certain formal properties of grammars. *Information and Control 2*(2), 137 – 167.

Dempster, A. P., N. M. Laird, and D. B. Rubin (1977). Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society, Series B 39*(1), 1–38.

Francis, W. N. (1964). *A standard sample of present-day English for use with digital computers*. Cooperative research project. Brown University.

Kasami, T. (1965). An efficient recognition and syntax-analysis algorithm for context-free languages. Technical report, Air Force Cambridge Research Lab, Bedford, MA.

Lari, K. and S. Young (1990). The estimation of stochastic context-free grammars using the inside-outside algorithm. *Computer Speech & Language 4*, 35–56.

Marcus, M., B. Santorini, and M. Marcinkiewicz (1993). Building a large annotated corpus of english: The penn treebank. *Computational Linguistics 19*(2), 313–330.

Prescher, D. (2001). Inside-outside estimation meets dynamic em. In *Proceedings of the 7th International Workshop on Parsing Technologies (IWPT-01), October 17-19. Beijing, China*, pp. 241–244.

Prescher, D. (2002). *EM-basierte maschinelle Lernverfahren für natürliche Sprachen*. Phd-thesis, Universität Stuttgart, Institut für Maschinelle Sprachverarbeitung (IMS), Stuttgart.

Prescher, D. (2005). A tutorial on the expectation-maximization algorithm including

maximum-likelihood estimation and em training of probabilistic context-free grammars. Technical report, Institute for Logic, Language and Computation, University of Amsterdam.

Stüber, T. (2012). Consistency of probabilistic context-free grammars. Technical report, Faculty of Computer Science, Technische Universität Dresden.

Younger, D. H. (1967). Recognition and parsing of context-free languages in time n3. *Information and Control 10*(2), 189 – 208.

Zhou, G. and K. Lua (1998). Training of probabilistic context-free grammar. Technical report, Dept. of Computer Science, National University of Singapore.

A Tags of the Penn Treebank

In this work, tags for syntactical categories and single words (latter also referenced to as part-of-speech, a terminology which was first introduced by Francis (1964)). A very common set of such tags is the one used by the Penn Treebank Project[1], a corpus of parse trees, which is further described by Marcus, Santorini, and Marcinkiewicz (1993).

The following list of tags is an excerpt of the whole list in the article of said authors:

S	Simple declarative clause
NP	Noun phrase
VP	Verb phrase
DT	Determiner
NN	Noun, singular or mass
NNP	Proper noun, singular
PRP	Personal pronoun
PP$	Possessive pronoun
VB	Verb, base form
VBD	Verb, past tense
VBP	Verb, non-3rd ps. sing. present

[1] http://www.cis.upenn.edu/~treebank/