

String Alignment in Grammatical Inference

what suffix trees can do

A thesis
submitted in partial fulfilment
of the requirements for the degree
of
Master of Arts
at
Tilburg University
by

Jeroen Geertzen

Supervised by :
Dr. M. M. van Zaanen

Computational Linguistics and AI
Faculty of Arts
Tilburg, The Netherlands

November 2003

Abstract

This thesis is concerned with unsupervised learning of syntactic structure from plain text corpora by aligning sentences. Based on Harris' (1951) linguistic notion of substitutability, sentences in a plain text corpus can be compared to each other and those parts that have similar context and in addition can be substituted for each other without resulting in ungrammatical sentences are considered to be possible constituents called hypotheses.

A system that uses such an approach is ABL (Alignment-Based Learning) [van Zaanen, 2000]. Currently, the main method used to align sentences and produce hypotheses is of such algorithmic complexity that ABL is feasible for small corpora only.

This thesis explores the main topics of unsupervised learning of syntactic structure of natural language and introduces new algorithms based on suffix trees. The algorithmic complexity of the new algorithms allow ABL to learn from large corpora as well. Furthermore, it shows that the suffix tree data structure has important advantages in finding regularities in corpora, which is a fundamental issue in many approaches to unsupervised and semi-supervised grammatical inference. The performance of the new algorithms with respect to a learning task are tested within the ABL framework on corpora of various sizes: the English ATIS, the Dutch OVIS, and the English Wall Street Journal treebank.

In conclusion, we observe that the time needed for alignment learning using the new algorithms as a function of the corpus size is indeed reduced in such an extent that learning on large sized corpora with systems like ABL is feasible. However, the recall of the hypothesis constituents introduced by the new algorithms is lower than that of the established algorithm and justifies further research.

Contents

Abstract	ii
List of Figures	vi
List of Tables	vii
1 Introduction	1
1.1 Preliminaries	3
1.1.1 Strings	3
1.1.2 Sentences and sentence related	4
1.1.3 Graphs and trees	4
2 Inducing syntactic structure	6
2.1 The learning and learnability of grammar	7
2.1.1 A definition of ‘learning’	7
2.1.2 Models of learning a grammar	8
2.1.2.1 Identification in the limit	8
2.1.2.2 PAC learning	9
2.2 Grammars and automata	11
2.3 Using likelihood with Probabilistic Context-Free Grammars	15
2.3.1 Maximum Likelihood	15
2.3.2 Inside-outside as training algorithm	19
2.3.3 Problems with learning PCFGs	19
2.4 Using compression (MDL)	20
2.5 Using distributional information	21
3 Comparing sentences	24
3.1 Edit distance	25
3.1.1 Introducing edit distance	25
3.1.2 Using edit distance to locate substitutable subsentences	29
3.2 Suffix trees	30
3.2.1 Introducing suffix trees	30
3.2.1.1 Tries	30
3.2.1.2 Suffix Trees	32

3.2.2	Ukkonen’s suffix tree algorithm	33
3.2.2.1	Suffix extension rules	34
3.2.2.2	Edge-label compression	35
3.2.2.3	Suffix links	35
3.2.2.4	Reducing suffix phases to $O(1)$	38
3.2.3	From strings to corpora	39
4	Learning by alignment	43
4.1	Hypotheses and the hypothesis space	43
4.2	Using edit distance	44
4.3	Using suffix trees	46
4.3.1	Where do hypotheses start	48
4.3.2	Closing hypotheses at the end of sentences	48
5	Empirical Results	55
5.1	ABL framework	55
5.2	Evaluating performance	57
5.3	Performance on the ATIS corpus	59
5.4	Performance on the OVIS corpus	61
5.5	Performance on the Wall Street Journal corpus	62
5.6	Performance on execution time	63
6	Conclusions and Future work	65
6.1	Conclusions	65
6.2	Future work	66
	Bibliography	67

List of Figures

2.1	Induction from the data produced by the target grammar	7
2.2	The error of g_h with respect to g_t	10
2.3	FSA corresponding to the grammar of Example 2.3	12
2.4	A nested structure that implies context-freeness of natural language	14
2.5	Cross-serial dependencies in Swiss-German	14
2.6	Two PCFGs that are able to produce $U = [acb, aacbb, aaacbbb]$	17
2.7	The two possible phrase structure trees corresponding with derivations for s given g_1	18
3.1	Edit transcript of <i>space</i> and <i>party</i>	25
3.2	Traces when substitution cost is 1 (left) and 2 (right)	28
3.3	Edit transcripts for two strings when substitution cost is 1 (left) and 2 (right)	29
3.4	Edit transcript for two sentences	29
3.5	A trie for $T = \{abc, aba, bd, cca, ccb\}$	31
3.6	A non-compact <i>suffix trie</i> (left) and a compact <i>suffix trie</i> of $S = \text{mimic}$	31
3.7	Suffix tree for $S = abac$	32
3.8	Implicit suffix trees for each suffix except ε in $S = abac$	33
3.9	The left tree is a fragment of a suffix tree with explicit edge labels whereas the right tree shows pointers to positions in $S = abcdefghabcdijkl$	36
3.10	STree(S) for $S = babac$ with suffix links (1, 5) and (5, root).	36
3.11	Implicit STree(<i>abacab</i>).	39
3.12	A suffix tree for $S = \text{de man en de kat}$ with word identifiers	40
3.13	A suffix tree for <i>de man % de kat \$</i>	41
3.14	Split operation in joint suffix tree for $S_1 = \text{de hond}$, $S_2 = \text{de kat}$	42
4.1	Links between common words of two sentences	45
4.2	Corpus U_1 and STree(U_1)	47
4.3	Sentence-position pairs before and after split operation	48
4.4	Corpus U_2 and STree(U_2)	50
4.5	Corpus U'_2 and STree(U'_2)	50
4.6	Matching opening and closing brackets in hypotheses generation.	52
5.1	Treebank fragment for the sentence <i>I want a flight from Ontario to Chicago</i>	57
5.2	Evaluating a learned treebank	58

5.3	Evaluation with ABL	59
5.4	Execution time for edit distance and suffix tree methods	64

List of Tables

- 2.1 Chomsky’s hierarchy of formal grammars 13
- 3.1 Suffix extension rules to be applied when building up $S\mathit{Tree}(abac)$ 35
- 5.1 Results for alignment learning on the ATIS corpus 60
- 5.2 Number of hypotheses after alignment learning on the ATIS corpus 60
- 5.3 Results for alignment learning on the OVIS corpus 61
- 5.4 Number of hypotheses after alignment learning on the OVIS corpus 61
- 5.5 Results for alignment learning on section 23 of WSJ 62
- 5.6 Number of hypotheses after alignment learning on section 23 of WSJ 62
- 5.7 Results for alignment learning on WSJ section 2-21+23 63
- 5.8 Number of hypotheses after alignment learning on WSJ section 2-21+23 63
- 5.9 Timing results (in seconds) for edit distance alignment and suffix tree alignment on several corpora 64

Chapter 1

Introduction

The process of learning the syntactic structure of a natural language is something that seems obvious for humans. However, when we want computers to learn to recognize and produce sentences that are grammatically correct, we need to specify two algorithms. One that is capable of inferring the grammar of a language from sentences that belong to this language and one that is capable of using the grammar to produce well formed sentences.

Just as young children do not acquire knowledge of the correct order of words by giving them a textbook on grammar, we would like a computer to learn syntax without specifying what results are correct and what results are not beforehand. In other words, we would like a computer to learn syntax in an unsupervised way.

To learn the syntactic structures of a language in an unsupervised way, different approaches and algorithms have been proposed. One group of algorithms use the minimal description length (MDL) principle that originates from information theory to describe the sentences that are presented with the minimum number of bits. Examples of algorithms that belong to this group are presented in [Wolff, 1988; Grünwald, 1996; de Marcken, 1996]. Another group of algorithms uses statistical principles such as maximum likelihood (ML) and Bayesian inference to infer a grammar that describes the language best. An example of such an algorithm is proposed in [Stolcke, 1994]. Some other algorithms make use of distributional information in inferring the grammar. Some examples are those presented in [Adriaans, 1999; Klein and Manning, 2001; Clark, 2001; van Zaanen, 2000]¹.

Two systems that make use of distributional information, EMILE, presented in [Adriaans, 1999] and Alignment-based Learning (ABL), presented in [van Zaanen, 2000] consider sentences that have a word or a concatenation of words in common. EMILE makes use of rule induction by clustering. ABL looks for parts of sentences that can be successfully substituted for each other and assumes them constituents of the same type. In this thesis, we will focus on ABL and look how this system is able to infer syntactic structure.

¹Whereas Adriaans and van Zaanen choose for a symbolic approach, Klein and Manning pursue a probabilistic approach using probability models over trees

ABL looks for interchangeable parts between the sentences in a pair and assumes interchangeable parts to share the same constituent type. Because the comparison and alignment of strings is central in ABL, the method that is used is important. One well-known metric that is used in ABL to align sentences in order to find the substitutable parts is the *Levenshtein distance* [Levenshtein, 1965]. This metric, which is employed in various applications such as spelling checkers, is also known as *edit distance* and is formalized in [Wagner and Fischer, 1974]. Although the computation of edit distance in sentence pairs to find the substitutable parts works well, it is currently computationally costly in such extent that it is not feasible for ABL to learn from large² corpora. An alignment method that is less costly would allow ABL to learn from large corpora as well. This brings us to the motivation for this work: finding an *efficient* and *effective* method to infer regularities in linguistic corpora by aligning sentences.

In this thesis, we aim to give an introduction in the principles that underly different kinds of algorithms for (unsupervised) grammatical inference (Chapter 2). We are interested in the ways to find substitutable parts in sentence pairs, how the Levenshtein distance can be used for this purpose and how suffix tree can play a role (Chapter 3). In Chapter 4 we see how the edit distance can be used to align the sentences and to produce hypotheses for constituent pairs. The suffix tree data type introduced in Chapter 3 will be adjusted to allow a different method of hypothesis generation than the one that is based on the edit distance. To see if suffix tree based alignment can be an alternative for edit distance based alignment, we will compare both approaches within the ABL framework and present the experimental results in Chapter 5. In the last chapter we will reflect and conclude on the findings in the previous chapters.

Throughout the thesis, example sentences will be used to illustrate several concepts. These example sentences will be alternately in Dutch and English. In the former case, English translations will be provided. The choice of using multilingual examples reflects the desire to focus on algorithms that can learn syntax of a language without requiring any language dependent information in the learning process.

To conclude this introductory section, I would like to thank some people who contributed directly or indirectly to this thesis. Menno van Zaanen, my supervisor, for his valuable advice, guidance and computer time on his workstation. Riny Huybregts for explaining the insufficiency of CFGs for natural language modeling. Harry Bunt and Antal van den Bosch for discussing several ideas for a graduation theme. Roser Morante and Yann Girard for allowing me to work in their working space when I was in need of a nice place to work. Rob Freeman for his references to research in linguistics and machine learning. Above all I would like to thank my parents for their support during my course of studies.

²Say > 100K sentences.

1.1 Preliminaries

In this section we will establish the notations and definitions that will be used throughout the whole thesis. We do not attempt to define the concepts that will be used *ad fundum*, but key concepts will be explicitly defined according to their common interpretation in literature. In this section, (implicitly defined) mathematical objects are printed in *italic*. Following standard notation, braces $\{\}$ encapsulate sets, and parentheses $()$ surround ordered pairs. Where we want to express the complexity of algorithms, we use the notation commonly used in computer science.

1.1.1 Strings

Let N be the set of integers. Let $\{n, m, i, j\} \in N$.

Definition 1.1 (alphabet). An *alphabet* is a non-empty, finite set of symbols, denoted by Σ . It does not contain elements that can be formed from other elements of Σ .

Definition 1.2 (language). For a given alphabet Σ , a *language* over Σ is a subset of Σ^* . The subset \emptyset is called the *empty language*.

Definition 1.3 (string). An element of Σ^* is called a *string*.

For $w \in \Sigma^*$, $w = x_1x_2 \dots x_n$, where $x_i \in \Sigma$ and $1 \leq i \leq n$. The *length* of the string w , denoted by $|w|$, is $|w| = n$. If $n = 0$, w is said to be the *empty string*. It is denoted by ϵ , that is, $|\epsilon| = 0$. The i -th symbol of w is denoted by $w[i]$ where $1 \leq i \leq n$.

Definition 1.4 (concatenation). Let x, y be strings and $x = x_1x_2 \dots x_m$, $y = y_1y_2 \dots y_n$ where each x_i and x_j is in Σ and $1 \leq i \leq m$ and $1 \leq j \leq n$. Then, $x_1x_2 \dots x_my_1y_2 \dots y_n$ is said to be the *concatenation* of string x and string y and is denoted by $x \cdot y$, or simply xy .

Definition 1.5 (substring). If $x, y, z \in \Sigma^*$ and w is the string $w = xyz$, then y is said to be a *substring* of w . If at least one x, z is different from ϵ , then y is called a *proper substring*. When the substring y is non-empty, and $n = |y|$, y can be denoted in the context of w by $w[i..j]$ where y starts at position i , that is, $w[i] = y[1]$ and y ends at position j , that is $w[j] = y[n]$.

Alternatively, y can be denoted in the context of w by $y_{i..j}^w$ with $i \leq j$ where y is a list of $j - i$ elements of w where for each k with $1 \leq k \leq j - i$: $y_{i..j}^w[k] = w[i + k]$. In this notation, a substring y is empty when $i = j$ and span the entire sentence w when $i = 0$ and $j = |w|$.

Definition 1.6 (suffix). A *suffix* of string S of length $n = |S|$ is the empty string when $n = 0$ or else a substring of S that begins at position i where $1 \leq i \leq n$ and ends at position n . The set of suffixes of a string S is denoted as $Suffix(S)$.

Definition 1.7 (prefix). A *prefix* of string S of length $n = |S|$ is the empty string when $n = 0$ or else a substring of S that begins at position l and ends at position i where $1 \leq i \leq n$. The set of prefixes of a string S is denoted as $Prefix(S)$.

1.1.2 Sentences and sentence related

When addressing sentences in natural language, the set theory of strings will be used as defined in Section 1.1.1 and words are assumed to be the symbols in the alphabet.

Definition 1.8 (corpus). A *corpus* U of size $n = |U|$ is a non-empty list of strings $[S_1, \dots, S_n]$.

Definition 1.9 (constituent). A *constituent* in string w is a tuple $c^w = \langle y, n \rangle$ where y is a substring of w and n is the non-terminal of the constituent and is taken from the set of non-terminals.

1.1.3 Graphs and trees

A graph consists of nodes and edges. Edges are used to connect two nodes. Nodes are generally denoted using letters u and v , and edges with e . A special kind of graph is the *directed graph*, which has restrictions on edges to allow hierarchical, parent-child relationships possible between nodes in the graph. More formally, we can define:

Definition 1.10 (directed graph). Let V be a finite set of *nodes*: $V = \{v_1, v_2, \dots, v_n\}$. Let E be a finite set of *edges*: $E = \{e_1, e_2, \dots, e_p\}$. Each edge $e_k \in E$ is of the form $e_k = (v_i, v_j)$ where $v_i, v_j \in V$. Then $G = (V, E)$ is said to be a *directed graph*.

In the edge $e_k = (v_i, v_j)$, v_i is called the *parent of* v_j and v_j is called the *child of* v_i and e_k is a *directed edge* from v_i to v_j . The edge e_k is said to be an *out-going* edge of v_i and an *in-coming* edge of v_j .

The number of in-coming (resp. out-going) edges of a particular node u_k is said to be the *in-degree* (resp. *out-degree*) of u_k .

Intuitively, we can define a *path* to be a sequence of nodes in a particular graph connected by edges. The *length* or *depth* of a path is the number of edges it takes from start node to end node.

Definition 1.11 (path). In a directed graph $G = (V, E)$, the sequence of nodes u_0, u_1, \dots, u_n is called a *path* if $(u_{i-1}, u_i) \in E$ for each $i(1 \leq i \leq n)$. The *depth* of the path is n .

A path with $u_0 = u_n$ is called a *cycle*. If there are no cyclic paths in G , the graphs is called a *directed acyclic graph (DAG)*.

Definition 1.12 (tree). A directed graph G is called a *tree* when it satisfies the following two conditions:

1. There exists exactly one node in G with in-degree= 0, known as the *root node*;
2. For each node u in G there exists exactly one path from the root node to u .

Any node in a tree other than the root node has exactly one parent node. Nodes with out-degree= 0 are called *leaf nodes*. A node that is neither root node nor leaf node is called an *internal node*. If there exists a path from node u to node v , u is said to be an *ancestor* of v , and v is said to be a *descendant* of u .

Chapter 2

Inducing syntactic structure

Generally speaking, the problem of learning a natural language can be divided in several subproblems. Among others, the learner needs to be able to segment the linguistic input in meaningful parts and lexical items. There is the acquisition of the syntax of a language (i.e. rules for recognizing and generating valid sentences in the language) and its semantics (i.e. the underlying meaning conveyed by the sentences). The learning of the syntax of the language is usually referred to as *grammatical inference* or *grammar induction*. The product of this process is a *grammar*, a formalism that captures the syntax of a language.

Why should we want to induce syntactic structure from natural language? One answer might be *to model the acquisition of human language*. Since we can observe regularities, and more strongly, syntactic structure in natural language it would be plausible to ask ourselves how this syntactic structure relates to the processes in our brain which enables us to acquire and use language. This kind of perspective would make us consider the induction of syntactic structure from a cognitive and psycholinguistic point of view. Another answer could come from a point of view of engineering. For applications like speech recognition, grammar checkers, but also the acquirement of meta data for linguistic research, it would be interesting to be able to induce the regularities that can be observed in natural language *computationally*. I.e., the process of induction can be defined precisely by procedures that can be implemented and executed in the form of a computer program. With the latter point of view, we will take a closer look to grammatical inference and restrain ourselves from any cognitive issues. Although a computational approach does not necessarily implies the use of a computer or more generally, a machine, we will consider grammar induction in the context of machine learning.

This chapter will give an overview of the problem of computationally learning the grammar of a language. It will describe the major approaches that have marked this field of study and it will introduce the necessary concepts to place these approaches into context.

2.1 The learning and learnability of grammar

2.1.1 A definition of ‘learning’

Let us first have look at a straightforward definition of what it is for a computer program to learn something. Mitchell [1997] gives a definition that introduces three elements: a *task* or a class of tasks, a *measure of performance* and a *source of experience*.

Definition 2.1. A computer program is said to **learn** from experience E with respect to some class of tasks T and performance measure P , if its performance at tasks in T , as measured by P , improves with experience E .

When we use this general definition we can say that the task is to learn a grammar, the performance measure could be a metric that calculates the difference between the grammar found and the target grammar (i.e., the grammar to be learned) and the experience could be the linguistic input in one or another form (e.g. plain text sentences, parse trees).

We can define a collection of grammars that are possible to learn. We call this collection the *hypothesis space*. One of these grammars is the grammar that the learning algorithm is supposed to learn (the *target grammar*)¹. Then, we could say that an algorithm for grammar induction typically should identify an *hypothesis grammar* g_h from the hypothesis space \mathcal{G} as the *target grammar* g_t . Note that $g_t, g_h \in \mathcal{G}$. The process of induction can then be depicted in its context in Figure 2.1.

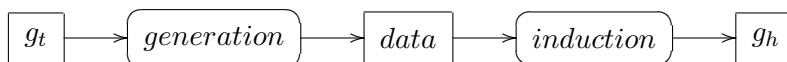


Figure 2.1: Induction from the data produced by the target grammar

In the process of induction, the algorithm can use different kinds of information. This information can be the knowledge whether a sentence of the input is a *positive training example* or a *negative training example*. In case of a positive training example, the sentence belongs to the language defined by the target grammar and is therefore called ‘grammatical’. Negative training examples are sentences not belonging to the language defined by the target grammar and are called ‘ungrammatical’.

One of the problems in identifying g_t with g_h is that in one way or the other, we need to compare the grammars and see if they are the same. We can do this indirectly by comparing the languages they produce². Let us take a set of positive example sentences P and a set

¹Here we implicitly assume that a target grammar for a natural language exists. In other words, we assume that all regularities in such a language can be captured in one formalism. We also assume that the target grammar is within the hypothesis space (otherwise the learner can never identify the target grammar).

²In Section 2.2 on page 11 this problem is addressed briefly.

of negative example sentences N such that $P, N \subset \Sigma^*$ and $P \cap N = \emptyset$. Let us define the binary predicate $cover(x,y)$ which is true when a grammar x can produce sentence y . Under the most ideal circumstances, the learner produces g_h such that:

$$\forall p \in P : covers(g_h, p) \quad (2.1)$$

$$\neg \exists n \in N : covers(g_h, n) \quad (2.2)$$

When (2.1) is true, we call g_h *complete* and when (2.2) is true, we call g_h *consistent*. Conversely, when (2.1) is false, we call g_h *incomplete* and when (2.2) is false, we call g_h *inconsistent*. Thus, g_h should be both complete and consistent.

In the kind of information that is available to the learner, we can observe a dichotomy that characterizes the learning process: *supervised* and *unsupervised learning*. In an arbitrary classification task, we speak of supervised learning when the learner has the availability of the correct classification of each example. In the context of grammar induction, we speak of supervised learning when the learner has availability of information on the grammar that is supposed to be inferred. As such, knowledge of whether the training example is positive or negative does not make the induction of grammar supervised whereas partial bracketing with constituent information does. Hence, supervised learning in the context of grammar induction often involves a pre-labeled corpus, where the inference algorithm can learn by the supervision of the labels. In the rest of this thesis we will concentrate on unsupervised learning only.

2.1.2 Models of learning a grammar

In succession of a general definition of what ‘learning’ is, we introduce two frameworks for learning a grammar which are frequently mentioned in literature. The frameworks characterize the notion of learning and learnability and address the conditions under which successful or unsuccessful learning is possible.

2.1.2.1 Identification in the limit

Early substantial research is that of Solomonoff [1964] and Gold [1967]. Gold proposed a model of on-line, incremental learning that he coined *identification in the limit* and simultaneously proposed a definition of ‘learning a language or grammar’.

Gold defined the learning process in the following way. After each time t the learner receives a new example i_t , the learner must return some hypothesis $H(i_1, i_2, \dots, i_t)$ and the target language is identified in finite time at the moment the learner returns a hypothesis that matches the target language and does not change its mind when processing other examples. At that moment, a class of languages in which the target language takes part is said to be

identifiable in the limit. It is important to note that in the case of identifiability in the limit, the learner does not need to know when his hypothesis is correct. Or, as put by Gold:

“My justification for studying identifiability in the limit is this: A person does not know when he is speaking a language correctly; there is always the possibility that he will find that his grammar contains an error. But we can guarantee that a child will eventually learn a natural language, even if it will not know when it is correct.” [Gold, 67, pp. 450]

Gold introduces two kinds of information presentation: *text* and *informant*. In the former case, only positive examples of the target language are presented. In the latter case, both positive and negative examples are presented and an informant can tell the learner whether any string belongs to the target language. Important in this context is that Gold proved that the classes of languages involved in natural language, such as context-sensitive, context-free and regular languages, are not identifiable in the limit when using the *text* method of information presentation. In other words: only using positive examples is insufficient. Thus, learning a language (e.g. context-free) according to Gold’s definition of learnability is not possible unless the learner has *both* positive and negative examples.

To illustrate the importance of negative examples in learning, for instance, a context-free target grammar, consider a hypothesis grammar that is a superset of the target grammar. For instance, let us assume a hypothesis grammar for English which is utterly simple and constructs sentences by concatenating words from the lexicon. Furthermore, let us assume that English has a fixed number of words in its lexicon. If we have only positive examples there is nothing wrong with the hypothesis grammar since it generates all the positive examples. However, at the same time the hypothesis grammar also produces many sentences which are ungrammatical, but without negative examples there is no way of knowing the degree of consistency of the grammar.

Gold’s conclusion about the need of negative examples is somewhat contradictory to the observations by Brown and Hanlon [1970], who argue in conclusion of empirical results that children receive only positive evidence in language acquisition. There are some researchers that have contested this conclusion and suggest that implicit and explicit evidence does appear with enough significance in the input children receive (see [Sokolov and Snow, 1994] for a review). This direction of research will not be dealt with any further because it would bring us from an exclusively computational point of view to cognitive issues.

2.1.2.2 PAC learning

Valiant [1984] proposed an alternative learning process for identification in the limit, called *Probably Approximately Correct* (PAC) learning. Whereas identification in the limit assumes finite time and requires exact learning, PAC learning poses a polynomial time com-

plexity constraint and does not require exact learning but allows g_h to identify g_t with a certain probability.

Let $X = \Sigma^*$ refer to the set of all possible sentences given a certain alphabet. Let us define an unknown probability distribution D over X . Example sentences are generated by drawing a sentence $x \in X$ according to D and presented to the learner together with the knowledge whether it occurs in $L(g_t)$ or not. After a sequence of examples the learner must output some hypothesis grammar g_h from the hypothesis space as estimate of g_t . Because we are interested in how closely the output of the learner, g_h , approximates the target grammar, g_t , we define an error metric for the hypothesis grammar:

Definition 2.2 (error). The *error*, denoted with $Error_D(g_h)$, of the hypothesis grammar g_h with respect to the target grammar g_t is the probability that g_h and g_t disagree on the classification of randomly drawn instances x from distribution D . When we define $covers(a,b)$ to return the value 1 when grammar a can produce sentence b and to return the value 0 when grammar a cannot produce sentence b , we can define:

$$Error_D(g_h) = P_{x \in D}(covers(g_t, x) \neq covers(g_h, x))$$

Figure 2.2 shows this definition in graphical form. When $Error_D(g_h) = 0$, g_h and g_t are

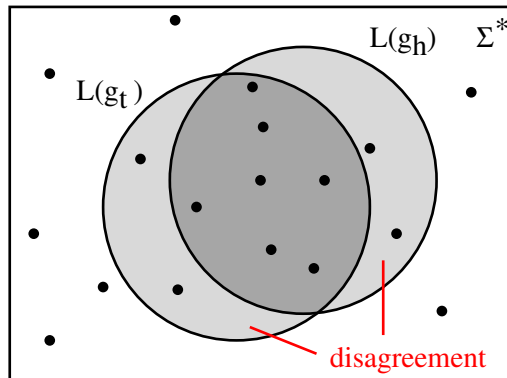


Figure 2.2: The error of g_h with respect to g_t

said to be identical. However, this is unlikely to happen because then we need to provide examples for every possible instance of X in order to rule out the possibility of multiple hypotheses that are consistent with the example sentences. In order to select an *approximately correct* hypothesis we allow a non-zero error that is bound by some constant, ϵ :

$$Error_D(g_h) < \epsilon$$

Because the example sentences are drawn randomly, there will still be a non-zero probability that the sequence of training examples will be misleading to the learner. We will cope with

this by taking into account a *probability* of failure that is bounded by some constant δ .

$$P[\text{Error}_D(g_h) < \epsilon] > (1 - \delta)$$

Hence, we have come to a *probably approximately correct* learning process. The learner must output with sufficient high probability $(1 - \delta)$ an hypothesis that has a sufficient low error (ϵ). To complete the framework of PAC-learning, we need to restrict the computational resources to polynomial complexity and can define PAC-learnability as in Definition 2.3. Since it makes sense to define this kind of learnability for grammar classes rather than a single target grammar, we define a grammar class C such that C is a subset of the hypothesis space of all grammars, \mathcal{G} , i.e. $C \subseteq \mathcal{G}$.

Definition 2.3 (PAC-learnability). Let us take C to be defined over a set of example sentences from X of length n . C is *PAC-learnable* by the learner if for all grammars $g \in C$, distributions D over X , ϵ and δ with $0 < \epsilon, \delta < 0.5$, the learner will with $P > (1 - \delta)$ output a hypothesis grammar g_h with $g_h \in \mathcal{G}$ such that $\text{error}_D(g_h) < \epsilon$, in time that is polynomial in $\frac{1}{\epsilon}$, $\frac{1}{\delta}$, n , and in the size of g .

Note that this definition requires the existence of a hypothesis with small enough error for every target grammar in the class C when we want to show that C is PAC-learnable. If one of the grammars in the class does not meet the requirements given, the whole class is not PAC-learnable.

One of the drawbacks of PAC-learnability as defined by Valiant [1984] is that learning should take place under any distribution. This requirement is in many cases too stringent, allowing unrealistic and unnatural examples. For instance, it would not be unrealistic to provide the learner with simple examples first in order to learn a certain concept. For this reason, Li and Vitányi [1991] proposed a model for PAC-learning with simple sentences called simple PAC learning (abbreviated with PACS). In this model, the class of distributions is restricted to simple distributions only.

2.2 Grammars and automata

Although the notion of *grammar* has been informally introduced and used occasionally, we need a formal definition. Up to now, a grammar has been described as a formalism that captures the regularities and constraints on word order of a language. Fundamental aspects in traditional grammars is that actual words are grouped in word classes known as *syntactic categories* or *parts of speech* (POS) which are represented in the grammar by symbols. In a language, these parts of speech can reoccur in different groups, called *constituents*.

From the early research on grammatical inference, *formal languages* were used to model natural languages. In these formal languages, symbols instead of words are used to represent

the elementary elements in a language. Now, let us formalize the notion of ‘grammar’.

Definition 2.4 (grammar). A formal grammar is a four-tuple $G = (V_T, V_N, P, S)$, where V_T is the set of terminal symbols (*terminals*), V_N is the set of *non-terminals*, P is the set of *production rules* for generating *valid* (i.e. ‘grammatical’) sentences of the languages. $S \in V_N$ is the *start symbol*. $V_T \cap V_N = \emptyset$. Elements of P are pairs (α, β) with $\alpha \rightarrow \beta$, meaning that α can be rewritten to β . Let $V = V_T \cup V_N$, then $\alpha, \beta \in V^*$ and α is non-empty.

Terminal symbols are symbols that appear in the final strings. Non-terminal symbols are symbols that are expanded into other symbols. Example 2.3 shows a small grammar.

$$(2.3) \quad G = (\quad \{a, b, c\}, \{S, B\}, P, S \quad)$$

$$\text{with } P = \left\{ \begin{array}{l} S \rightarrow aS \\ S \rightarrow cbbB \\ B \rightarrow bB \\ B \rightarrow b \end{array} \right\}$$

This grammar generates the language $L(G)$ as specified in Example 2.4 and a *derivation* of the string $aacbbbb$ from L is given in Example 2.5. A derivation is a sequence of productions that lead to a particular string. Thus, a *language* can be defined as all sentences that can be produced using the production rules of the grammar.

$$(2.4) \quad L(G) = \{ a^k cbb b^n \mid k \geq 0 \wedge n \geq 1 \}$$

$$(2.5) \quad S \Rightarrow aS \Rightarrow aaS \Rightarrow aacbbB \Rightarrow aacbbbB \Rightarrow aacbbbbbB \Rightarrow aacbbbbb$$

The formal languages can be associated with (imaginary) devices, called *automata*, that produce or recognize exactly this formal language. The task of grammatical inference can then be considered as inducing the correct automaton for producing or recognizing the target language. For example, the finite state grammar example corresponds to the finite state automaton in Figure 2.3.

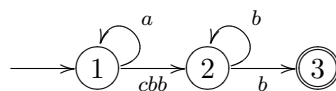


Figure 2.3: FSA corresponding to the grammar of Example 2.3

Based on the kind of restrictions that can be put on the production rules, Chomsky [1957] proposed a hierarchy of formal grammar types. These types are listed in Table 2.1, along with the corresponding automaton that produces or recognizes the language the grammar

Table 2.1: Chomsky’s hierarchy of formal grammars

Type	Name	Automaton
Type 3	Regular Grammars (RGs)	Finite State Automata (FSAs)
Type 2	Context-Free Grammars (CFGs)	Push-Down Automata
Type 1	Context-Sensitive Grammars (CSGs)	Linear Bounded Automata (LBAs)
Type 0	Unrestricted Grammars	Turing Machines

describes. The hierarchy that Chomsky proposed can be called arbitrary in the sense that there are alternative ways to put formal grammars in a hierarchy but since Chomsky’s hierarchy turned out to be practical in its use, it is frequently used in the terms of formal grammars. The hierarchical relation is described by:

$$type_3 \subset type_2 \subset type_1 \subset type_0$$

which expresses that the set of languages produced by a type 3 grammar is a proper subset of the set of languages produced by a type 2 grammar et cetera. Finite state grammars, better known as *regular grammars*, have the limitation on the production rules that they cannot memorize a sequence over multiple production rules: they can only be either left-linear or right-linear. Left linear grammars use rules that always end with a terminal symbol: the rules appear in the form $A \rightarrow a$ and $A \rightarrow Ba$. Right linear grammars use rules that always begin with a terminal symbol: the rules appear in the form $A \rightarrow a$ and $A \rightarrow aB$. These restrictions make that a regular grammar could not, for instance, produce palindromes. However, when we extend the automaton for a regular grammar, a finite state automaton, with a push-down stack, we get a push-down automaton and we can produce or recognize the language a context-free grammar describes. A *context-free* grammar has production rules that appear in the form $A \rightarrow \beta$, where $A \in V_N$ and β is a sequence of one or more symbols from the union set $V = V_N \cup V_T$ or the empty string (ϵ). *Context-sensitive grammars* are more powerful and the production rules appear in the form $\alpha \rightarrow \beta$, where α and β represent arbitrary sequences of symbols of V and where $|\alpha| \leq |\beta|$. These restrictions allow production rules that have the following form: $\alpha_1 A \alpha_2 \rightarrow \alpha_1 \beta \alpha_2$, which can be read as $A \rightarrow \beta$ in the context of $\alpha_1 \cdots \alpha_2 P$. The last type of grammars in the hierarchy are the *unrestricted grammars*, sometimes called *recursively enumerable sets*. Their production rules are of the form $\alpha \rightarrow \beta$ where $\alpha \in V^+$ and $\beta \in V^*$. Thus, allowing almost all rules imaginable.

In his work *Syntactic Structures*, Chomsky [1957] showed with recursivity and embedding that natural languages require at least context-free power. Chomsky demonstrated with success that English and other natural languages, a sentence may contain nested structures that, when infinitely nested, cannot be produced or recognized by a FS automaton. An example of a nested structure is given in Figure 2.4. He also suggested (but did not provide evidence) that stronger grammars are necessary, but may not require the full power of a context sen-

if we either give her chocolate or thee, then she will likely start smoking

Figure 2.4: A nested structure that implies context-freeness of natural language

sitive language. He proposed combining a CF grammar with a set of CS rewriting rules as a model of natural language syntax. Huybregts [1984] and Shieber [1985] showed that context-free languages are too weak, using *cross-serial dependencies* in Swiss-German. In such constructions, the NPs and the verbs that take them as objects occur in cross-serial order and there exists a syntactic dependency between the pairs of constituents based on case-marking. An example in Swiss-German³ is given in Figure 2.5. As a side-effect of

Jan	säit	das	mer	d'chind	em	Hans	es	huus	lönd	hälfe	aastrische
Jan	says	that	we	the children	Hans		the	house	let	help	paint

Figure 2.5: Cross-serial dependencies in Swiss-German

Chomsky's analysis that natural language is at least context-free, FS automata (as well as statistical approaches) moved to the background. Today, FS approaches are back in the center of research for several reasons. High-level grammar formalisms turned out not to be very successful and in some cases, like in the field of phonology, it turned out that FS models were sufficient for certain purposes. Although English as a whole cannot be produced by a FS grammar, there are subsets of English where a FS model is adequate. Examples are expressions of postal addresses, time and date. Another advantage of FS approaches is that they are very efficient and are well-understood. FSGs are, nevertheless, not suitable for describing *full* natural language syntax and we will not discuss them any further.

In the following three sections, we will briefly consider three kind of approaches to unsupervised learning of syntax of natural language, those that are based on likelihood, compression and distribution. Because of the focus on natural language, we will mainly be concerned with context-free languages since they come closer to fully describing natural language than finite state languages.

³Translates to English as: *Jan says we let the children help Hans paint the house.*

2.3 Using likelihood with Probabilistic Context-Free Grammars

Stochastic or probabilistic context-free grammars (PCFGs) are basically context-free grammars with a probability distribution over the production rules. At first glance, it seems that for the learning of grammar of natural language, CFGs are clearly preferable over PCFGs. After all, learning a PCFG requires to learn the CFG plus the probability distribution over the production rules of the CFG. However, due to this probability property, a PCFG can give an idea of the plausibility of the grammar with respect to the sentences to learn. Another advantage of the probabilistic approach is that it can handle some noise in the input. E.g., real data tends to have errors and grammatical mistakes.

One aspect that makes PCFGs particularly suitable for grammar induction is that the limitations that Gold introduced for learning with only positive example sentences do not hold strongly against using PCFGs. As we have discussed earlier, Gold [1967] showed that CFGs cannot be learned (identified in the limit) without the use of negative examples. However, Horning [1969] showed that PCFGs *can* be learned from positive examples alone.

Now, let \mathcal{G} be the hypothesis space of grammars that are consistent with the linguistic input. If we define the learning of the target grammar g_t as the search for the grammar g_h in the hypothesis space \mathcal{G} that is most likely to be equivalent with the target grammar, we need an evaluation metric. When we give the grammars in the hypothesis space a stochastic property, we are able to use *likelihood* as a metric together with Bayesian inference to favor one grammar over the other.

Likelihood can be defined as the hypothetical probability that an event which has already occurred would yield a specific outcome. Where a *probability* refers to occurrence of future events, likelihood refers to past events with known outcomes and can therefore be considered *a posteriori*.

2.3.1 Maximum Likelihood

Let us assume a corpus U of sentences that are produced by g_t and a set \mathcal{G} of possible grammars:

$$U = [s_1, s_2, \dots, s_n], \quad \mathcal{G} = \{g_1, g_2, \dots, g_m\}$$

We want to find the most probable hypothesis $g \in \mathcal{G}$ given the evidence U (a conditional probability). Any such hypothesis is called a *maximum a posteriori* (MAP) hypothesis.

$$g_{MAP} = \arg \max_{g \in \mathcal{G}} P(g|U) \tag{2.6}$$

How to calculate the posterior probability of A given B is given by *Bayes' theorem*:

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)} \quad (2.7)$$

By applying Bayes' theorem to the right-hand side of Equation 2.6, we can calculate the posterior probability of each candidate hypothesis and subsequently select the candidate hypothesis with the highest posterior probability:

$$\begin{aligned} g_{MAP} &= \arg \max_{g \in \mathcal{G}} \frac{P(U|g) \cdot P(g)}{P(U)} \\ &= \arg \max_{g \in \mathcal{G}} P(U|g) \cdot P(g) \end{aligned} \quad (2.8)$$

Notice that in the final step we dropped the term $P(U)$ because it is constant and independent of g .

Now we can consider how to use the maximum a posteriori hypothesis in preferring one PCFG over others. A PCFG is a four-tuple which can be defined similar to a general grammar as in Definition 2.4. In addition, each rule has a probability and the left-hand side of a rule consist of a single non-terminal (CF-condition).

From its definition, it should be clear that for PCFGs, each rule in the set of production rules P has a probability, $P(A^i \rightarrow \beta^j)$, such that:

$$\forall_i \sum_j P(A^i \rightarrow \beta^j) = 1 \quad (2.9)$$

where $A \in V_N$, $\beta \in V^*$, A^i indicates the i -th non-terminal. I.e., the probabilities of all the rules that expand the same non-terminal i must sum to 1.

The probability of a sentence s in a PCFG is the sum of the probabilities of all derivations of s in the PCFG. Although we have used the notion of derivation earlier in this thesis, we now give a more formal definition with Definition 2.5.

Definition 2.5 (left-most derivation). A *derivation* of a string s in a grammar G is a sequence of sequences v_1, \dots, v_k over V^* , beginning with the start symbol S , in which $v_k = s$ and in which a transition of string v_i to v_{i+1} corresponds to the rewriting of the left-most non-terminal in v_i resulting in v_{i+1} according to a production rule belonging to G . A derivation is denoted as: $S \Rightarrow \dots \Rightarrow v_k$.

Now we have defined a derivation, we can define the probability of a derivation, which is the product of the probabilities of the production rules in the consecutive transitions of the derivation. More formally, we can define the probability of a derivation $S \Rightarrow \dots \Rightarrow v_k$

recursively with the following equation.

$$P(S \Rightarrow \dots \Rightarrow v_k) = P(S \Rightarrow \dots \Rightarrow v_{k-1})P(\alpha \rightarrow \beta) \quad (2.10)$$

where $\alpha \rightarrow \beta$ is the production rule used in the transition from v_{k-1} to v_k . So, when d is a derivation and $D(s)$ is the set of all possible derivations for s , we have

$$\begin{aligned} P(s) &= \sum_{d \in D(s)} P(s, d) \\ &= \sum_{d \in D(s)} P(d) P(s | d) \end{aligned} \quad (2.11)$$

$$= \sum_{d \in D(s)} P(d) \quad (2.12)$$

where 2.12 follows from 2.11 on the condition that the terminal symbols are included in the production rules so that the probability of s given d is 1.

Let us take a look to a very simple example. For U and \mathcal{G} we have the following:

$$U = [acb, aacb, aaacbbb], \quad \mathcal{G} = \{g_1, g_2\}$$

where g_1, g_2 are two PCFGs specified in Figure 2.6. The question for the learner is which

S	\rightarrow	SS	$\frac{1}{2}$		S	\rightarrow	aSb	$\frac{1}{2}$
S	\rightarrow	a	$\frac{1}{5}$		S	\rightarrow	c	$\frac{1}{2}$
S	\rightarrow	b	$\frac{3}{5}$					
S	\rightarrow	c	$\frac{1}{10}$					

Figure 2.6: Two PCFGs that are able to produce $U = [acb, aacb, aaacbbb]$

grammar to prefer given U . Without considering any probabilistic information we could intuitively predict g_2 to be preferred over g_1 because all the sentences in U clearly follow the pattern $\{a^n c b^n \mid n \geq 1\}$. To illustrate the use of likelihood in the decision which grammar to prefer, we take sentence $s = acb$. In order to use the probabilistic information to determine which grammar is to prefer, we have to consider the derivations which are possible for s in both grammars and to compute the probability for the derivations. For g_1 we observe two possible derivations:

$$\begin{aligned} S &\Rightarrow SS \Rightarrow aS \Rightarrow aSS \Rightarrow acS \Rightarrow acb \\ S &\Rightarrow SS \Rightarrow SSS \Rightarrow aSS \Rightarrow acS \Rightarrow acb \end{aligned}$$

which could be visualized by the trees in Figure 2.3.1.

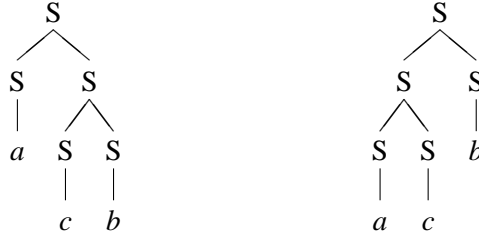


Figure 2.7: The two possible phrase structure trees corresponding with derivations for s given g_1

For g_2 we observe a single possible derivation:

$$S \Rightarrow aSb \Rightarrow acb$$

Now, when we take sentence $s = acb$, each derivation in g_1 (t_{g_1}) will have the probability $P(t_{g_1})$ and each derivation in g_2 (t_{g_2}) will have the probability $P(t_{g_2})$ ⁴.

$$P(t_{g_1}) = \left(\frac{1}{2}\right)^2 \cdot \left(\frac{1}{5}\right)^2 \cdot \frac{1}{10} = \frac{1}{1000} \quad (2.13)$$

$$P(t_{g_2}) = \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4} \quad (2.14)$$

In g_1 there are two possible derivations and in g_2 just one possible derivation for s . Taking this into account we can calculate for both grammars the probability for s , which we shall consider for the sake of simplicity the same as the calculation of $P(U|g)$ ⁵:

$$P(acb|g_1) = 2 \cdot P(t_{g_1}) = \frac{1}{500} \quad (2.15)$$

$$P(acb|g_2) = 1 \cdot P(t_{g_2}) = \frac{1}{4} \quad (2.16)$$

From Equation 2.15 and Equation 2.16 we can say that g_2 is indeed preferable over g_1 , because $P(U|g_2) > P(U|g_1)$. When we assume, for this little example, the prior probabilities to be equal, i.e. we assume $P(g_1) = P(g_2)$, then we can conclude that $g_{MAP} = g_2$. I.e., we have selected the grammar with the *maximum likelihood*.

Important in this example is to observe that PCFGs are biased in favor of smaller trees: non-terminals introducing a small number of rules will generally be favored over non-terminals introducing many rules since the probability of the rules that span under each non-terminal

⁴Note that the probability for each of the derivations in a particular grammar does not necessarily need to be the same.

⁵Here we simplify by taking the probability of s given the grammars and not the probability of U given the grammars. The latter involves more calculations (namely that of the other two sentences in U) but results in the same conclusion: $P(U|g_2) > P(U|g_1)$.

must sum to 1.

2.3.2 Inside-outside as training algorithm

Many researchers have also used the *Inside-Outside* (IO) algorithm in the learning of PCFGs.

The inside-outside algorithm is a re-estimation procedure for estimating the rule probabilities in a PCFG and was introduced by Baker [1979]. Given a set of positive example sentences and a PCFG of which the parameters are randomly initialized, the inside-outside algorithm first computes the most probable parse tree for each example sentence. The resulting derivations are then used to re-estimate all the probabilities associated with each rule as long as probability values are changing. This process is iterated until the optimal probabilities for the grammar rules are found. The assumption underlying this process is that a good grammar is one that makes the example sentences likely to occur. In other words, the process is looking for a grammar that maximizes the likelihood of the example sentences.

The results of this approach in [Pereira and Schabes, 1992; Carroll and Charniak, 1992] seem slightly discouraging in the sense that the inside-outside algorithm seems to converge to a local optimum (see Section 2.3.3). This, on its turn, means that the algorithm would almost always converge to a linguistically implausible grammar. More about inside-outside re-estimation can be found in [Baker, 1979; Charniak, 1993].

2.3.3 Problems with learning PCFGs

In order to learn PCFGs, we could agree on a very simple method:

1. Generate all possible PCFG production rules
2. Assign to the production rules initial probabilities
3. Run the training algorithm on a corpus of positive examples
4. Remove rules with .0 probability

Any such algorithm has the problem that there is no upper bound on the number of production rules to generate. The number will be so large that training is impractical. Of course we can limit the number of possible rules, but then we have the problem where to put the limit for we want to avoid to exclude the (a priori unknown) desired rules among the enormous number of rules that are not desirable. One step towards solving this problem is to restrict the non-terminals to a finite set so the introduction of new non-terminals and rules that involve these non-terminals is restricted. With a finite number of non-terminals there is still no restriction on the number of rules since one can keep making rules with longer

and longer right-hand sides. To prevent this we also need to introduce a constraint on the rule-length.

The method that has been sketched so far has been pursued by Lari and Young [1990]. Pereira and Schabes [1992] describe an approach to further limit the number of rules. Like Lari and Young they start with all possible production rules but use bracketed examples where the brackets indicate a partial parse tree. They modified the inside-outside algorithm to eliminate rules that do not conform to the bracketing. This highly reduces the number of parses that are required in the inside-outside algorithm, but its disadvantage is the lack of precisely bracketed corpora available. Another approach is to combine statistics and linguistic knowledge in order to reduce the number of rules.

A more serious problem for learning PCFGs is discussed by Carroll and Charniak [1992]. It appears that the training algorithms used for inferring the grammar (the inside-outside algorithm or equivalents) is converging in almost all experiments towards local optima. That is, in the hypothetically perfect circumstances (i.e. availability of all possible rules, a sufficient representative corpus) the algorithm easily manages to present a bad set of rules because it found some local optimum instead of the global optimum.

2.4 Using compression (MDL)

Learning a grammar obviously concerns a process of generalization. The production rules in the grammar are generalizations over the utterances or sentences of a language. Generally, the grammar and a lexicon are more compact than a full size corpus. In this context, generalization can be seen as compression. We can see the set of regularities that can compress the data the most is the optimal model that the learner should look for. A celebrated principle in science is Occam's razor, a popular inductive bias that can be summarized as 'choose the shortest explanation for the observed data'. When we combine this principle, which is more or the less formally described in information theory, with the power of Bayesian inference, we can formulate a principle that is called the *Minimum Description Length* (MDL) principle.

To describe MDL, we need to introduce a measure from information theory called *entropy*, which characterizes the (im)purity of an arbitrary collection of examples. Shannon and Weaver [1949] showed that the optimal code needed to encode a randomly drawn message i that has a probability of p_i requires at least $-\log_2 p_i$ bits. We will refer to the number of bits required to encode the message using code C as the *description length of the message with respect to C*. Given a collection S , containing samples which can be classified according to

c different values, then the entropy of S relative to this c -wise classification is defined as:

$$Entropy(S) \equiv \sum_{i=1}^c -p_i \log_2 p_i \quad (2.17)$$

where p_i is the proportion of S belonging to class i .

Now let us consider the equation of g_{MAP} we introduced in the previous section (Equation 2.8) and reintroduce it.

$$g_{MAP} = \arg \max_{g \in \mathcal{G}} P(S|g) \cdot P(g) \quad (2.18)$$

When taking the negative logarithm of the left-hand side, we get 2.19.

$$g_{MAP} = \arg \min_{g \in \mathcal{G}} -\log_2 P(S|g) - \log_2 P(g) \quad (2.19)$$

From Equation 2.19 we can conclude that short hypotheses are preferred, assuming a particular representation scheme for encoding hypotheses and data. We can use the *description length* to rewrite Equation 2.19 such that g_{MAP} is the hypothesis g that minimizes the sum given by the *description length of the hypothesis*, denoted by $l(g)$, plus the *description length of the data given the hypothesis*, denoted by $l(S|g)$.

$$g_{MDL} = \arg \min_{g \in \mathcal{G}} l(S|g) + l(g) \quad (2.20)$$

which is the *Minimum Description Length* (MDL) principle.

Thus, algorithms that use the MDL principle build grammars that describe the input examples with the minimum number of bits possible. An approach that uses MDL is described in [Grünwald, 1996]. Another theory, which is described by de Marcken [1996], uses the MDL principle to learn stochastic grammars from unsegmented text. Interesting to observe is that the theory is also used to attempt to learn directly from speech signals. Stolcke [1994] advocates the use of a Bayesian model selection criterion for HMMs and PCFGs but restricts it only to artificial languages. Wolff [1988] has argued for a notion on learning based on compression. Clark [2001] proposed combining distributional clustering (see next section) with MDL.

2.5 Using distributional information

A number of algorithms for grammar induction use *distributional* evidence to identify the constituent structure. The underlying assumption is that sequences of word or tags that are generated by the same non-terminal will appear in similar contexts and may be assumed to belong to the same constituent type.

One of the first attempts involved distributional clustering was by Lamb [1961]. Brill and Marcus [1992] consider the distributional similarity of part of speech tags to identify possible syntactic rules. In [Finch et al., 1995] some preliminary results show how distributional clustering algorithms can be used to find sets of tag sequences that occur in similar context. Their techniques produce some linguistically plausible clusters, but many implausible ones, and they do not demonstrate a complete grammar induction algorithm. Nevertheless, they show that distributional clustering can work with syntactic constituents. Klein and Manning [2001] present two distributional learning algorithms that use a probability model over trees and constituent context.

All the techniques mentioned so far use mainly *local* distributional context. There are also at least two approaches that use whole sentence contexts. Adriaans [1999] presents EMILE, which initially used a form of supervision but in later work [Adriaans et al., 2000; van Zaanen and Adriaans, 2001] is modified to be completely unsupervised. A similar approach is presented by van Zaanen [van Zaanen, 2000] with ABL. The algorithms make pairwise comparisons of sentences.

EMILE makes use of clustering of contexts and expressions. In the following example, the two example sentences have the same context.

<u>What is the</u>	time to leave	
<u>What is the</u>	best time to rest	
What is the	{time to leave best time to rest}	

If a group of contexts and expressions cluster together, they receive a type label which allows to form proto-rules of the form:

[0]	⇒	What is the	[12]
[12]	⇒	time to leave	
[12]	⇒	best time to rest	

Subsequently, EMILE will substitute expressions that are already clustered with the corresponding cluster label (see example below) and will use a proto rule when enough evidence is found.

<u>[64]</u>	⇒	think it is the right time to leave now
[64]	⇒	think it is the right [12] now

At the first glance, ABL seems not that much different. ABL makes use of contexts and expressions too, and groups (sub)sentences that are substitutable. Because of the substitutability of these parts they are considered of the same constituent type and receive a non-terminal symbol.

This [man]₁ walks

This [woman]₁ walks

However, where EMILE finds only a grammar rule where enough evidence is found, ABL stores all possible constituents and subsequently tries to select the best constituents. ABL uses Levenshtein distance [1965] to find substitutable parts in pairwise comparisons of sentences:

What is the []₂ time to [leave]₁

What is the [best]₂ time to [rest]₁

Where ABL only needs a pair of sentences with substitutable parts to learn structure, EMILE needs to have a certain support to establish rules. Because of this, EMILE needs larger corpora to learn whereas ABL can learn already on relative small corpora. In one-to-one comparison of the two systems [van Zaanen and Adriaans, 2001], EMILE appears to perform faster and less greedy than ABL. The results for both systems on the (relatively small) ATIS and OVIS corpora seem to be in slight favor of ABL.

Chapter 3

Comparing sentences

If we have sentences in a particular language and the grammar of this language is known, the syntactic structure of the sentences can be generated by using the grammar in the process of parsing the sentences¹. However, if the grammar is not known, we need to have a method that can generate the syntactic structure of sentences and does not assume grammatical knowledge to be available *a priori*. Methods that achieve this are generally called *structure bootstrapping systems*. In the process of generating the syntactic structure, a grammar is created implicitly and structure bootstrapping can be considered as a form of grammatical inference.

One of the methods to bootstrap structure is based on comparing sentences and grouping the unequal parts of sentences to the same constituent type when the remaining part of the sentences are the same. Actually, the unequal parts of sentences could be substituted in accordance with the notion of substitutability of segments in utterances Harris [1951] describes. In [van Zaanen, 2000], Harris' concept is used to design and implement an unsupervised (i.e. assuming no language-specific knowledge to be available) structure bootstrapping system. The key problem in this approach is to find the substitutable parts. This problem is explored in the next sections. We will have a look at two approaches that will be the main theme in this thesis: one using edit distance and one using suffix trees. For the sake of simplicity and compactness, we will start describing these approaches using simple strings over the English alphabet, to extend them later to the level of sentences. The only difference is in the alphabet; for strings it consists of characters whereas for sentences it consists of words. Likewise, we will refer to substrings of word-like symbols as subsentences.

Let us have a look to the exact matching problem and mention the important algorithms before introducing edit distance and suffix trees. When we want to find a substring S_{sub} in the string S , the most naive method to do this is to align the left end of S_{sub} with the left end of S , comparing the characters of S_{sub} and S pairwise until two unequal characters are found or until S is exhausted. In the latter case we have found a match. In either case

¹Even though the grammar is known this is not without any difficulty because ambiguities like PP-attachments will introduce multiple parses.

S_{sub} is shifted one place to the right and the process restarts until the right end of S_{sub} shifts past the right end of S . When we use n to denote the length of S and m to denote the end of S_{sub} , this naive method makes $O(nm)$ comparisons in the worst case. Many better algorithms can be found, ranging from relatively simple ones such as *n-position shift* to more complex exact string matching algorithms like the well known *Knuth-Morris-Pratt* [Knuth et al., 1977] and *Boyer-Moore* [Boyer and Moore, 1977] which solve exact matching in linear time ($O(m + n)$). The edit distance algorithm and the suffix tree algorithms that will be introduced are related to these algorithms in the sense that they both try to match strings. In addition, they also allow to find the parts that cannot be matched in the context of the parts that can be matched.

3.1 Edit distance

3.1.1 Introducing edit distance

The *edit distance* or *Levenshtein distance* [Levenshtein, 1965] can be defined as follows:

Definition 3.1 (edit distance). The edit distance between two strings s_1 and s_2 is the minimum cost needed to transform string s_1 into s_2 .

Transformation of one string in the other is possible by means of *edit operations*. Usually, three kind of edit operations are distinguished: insertion, deletion and substitution. Each kind of edit operation can have its own cost. Which operations are involved in a transformation from string x to string y are expressed in an *edit transcript*.

Definition 3.2 (edit transcript). An edit transcript is a list of labels denoting the edit operations needed to transform one string into another. These labels are E_i for insertion, E_d for deletion and E_s for substitution.

An illustrative example of an edit transcript can be found in Figure 3.1.

	E_d			E_s	E_s	E_i
string 1	s	p	a	c	e	-
string 2	-	p	a	r	t	y

Figure 3.1: Edit transcript of *space* and *party*.

The algorithm to calculate the edit distance makes use of a technique called *dynamic programming*, which has three components: the *recurrence relation*, the *tabular computation*, and the *traceback*. With the recurrence relation, we describe a recursive relationship between a solution and the solutions of its subproblems. We use the recurrence relation to fill

the cells of a matrix where each row represents a symbol of the first string and each column represent a symbol of the second string. The recurrence relation calculates the edit cost for each cell. The third component, traceback, finds those trace or traces in the matrix from the upper-left corner (the first symbol of both strings) to the lower-right corner (the last symbol of both strings) of the matrix for which the sum of the edit costs of the cells the trace visits is the lowest. The components which have been described briefly, will now be discussed in more detail.

For two strings S_1 and S_2 , $d(i, j)$ is defined to be the edit distance of $S_1[1..i]$ and $S_2[1..j]$. The recurrence relation is then defined as

$$d(i, j) = \min \left(\begin{array}{l} d(i-1, j) \quad + \quad c(S_1[i] \rightarrow \epsilon) \\ d(i, j-1) \quad + \quad c(\epsilon \rightarrow S_2[j]) \\ d(i-1, j-1) \quad + \quad c(S_1[i] \rightarrow S_2[j]) \end{array} \right) \quad (3.1)$$

where $d(0, 0) = 0$ and where c is the *edit cost function* $c(X \rightarrow Y)$ that returns a real value for substituting² X into Y . Thus, the right-hand term in the three additions in Equation 3.1 involve deletion, insertion and substitution respectively.

When no smaller indices exist, we need the base conditions given in 3.2 and 3.3.

$$d(i, 0) = d(i-1, 0) + c(S_1[i] \rightarrow \epsilon) \quad (3.2)$$

$$d(0, j) = d(0, j-1) + c(\epsilon \rightarrow S_2[j]) \quad (3.3)$$

With the recurrence relation and the base conditions, a $m \times n$ matrix can be created and filled where $m = |S_1|$ and $n = |S_2|$ according to an algorithm like the one in Algorithm 1, which resembles that of Wagner and Fischer [1974].

Now we have introduced the recurrence relation and the matrix, we can have a look at how they relate. From the left hand term in the three additions in Equation 3.1, we can see that deletion relates cells in the vertical direction of the matrix ($i-1$), insertion relates cells in the horizontal direction ($j-1$) and substitution relates cells in the diagonal direction ($i-1, j-1$). The final component in dynamic programming, the traceback, finds an edit transcript which results in the minimum edit cost. We give a simplified algorithm for the traceback in Algorithm 2. The traces found can be constructed alternatively by considering cell (m, n) in the matrix and look to the cost values in the cells $(m-1, n)$, $(m, n-1)$ and $(m-1, n-1)$. For the cell C with the lowest value, a part of a trace is fixed by linking cell (m, n) to cell C with a associated edit operation for this part depending on the orientation of C with respect to (m, n) . When multiple neighbouring cells have the same lowest value, a trace splits in multiple traces.

Now, let us take an example for the two strings $S_1 = abcd$ and $S_2 = abac$. When we

²Note that this use of the word has nothing to do with the edit operation 'substitute'. In this context, deletion and insertion can be define as substituting an empty symbol or a member of the alphabet.

Algorithm 1: Edit distance matrix for string S_1 and S_2

Require: S_1, S_2 : string
 i, j : integer

- 1: $d(0, 0) = 0$
- 2: **for** i **from** 1 **to** $|S_1|$ **do**
- 3: $d(i, 0) = d(i - 1, 0) + c(S_1[i] \rightarrow \epsilon)$
- 4: **end for**
- 5: **for** j **from** 1 **to** $|S_2|$ **do**
- 6: $d(0, j) = d(0, j - 1) + c(\epsilon \rightarrow S_2[j])$
- 7: **end for**
- 8: **for** i **from** 1 **to** $|S_1|$ **do**
- 9: **for** j **from** 1 **to** $|S_2|$ **do**
- 10: $d_{del} = d(i - 1, j) + c(S_1[i] \rightarrow \epsilon)$
- 11: $d_{ins} = d(i, j - 1) + c(\epsilon \rightarrow S_2[j])$
- 12: $d_{sub} = d(i - 1, j - 1) + c(S_1[i] \rightarrow S_2[j])$
- 13: $d(i, j) = \min(d_{del}, d_{ins}, d_{sub})$
- 14: **end for**
- 15: **end for**

Algorithm 2: Find a minimum edit cost transcript

Require: S_1, S_2 : string
 i, j : integer

- 1: $i = |S_1|$
- 2: $j = |S_2|$
- 3: **while** $i \neq 0$ **and** $j \neq 0$ **do**
- 4: **if** $d(i, j) = d(i - 1, j) + c(S_1[i] \rightarrow \epsilon)$ **then**
- 5: delete()
- 6: $i = i - 1$
- 7: **else if** $d(i, j) = d(i, j - 1) + c(\epsilon \rightarrow S_2[j])$ **then**
- 8: insert()
- 9: $j = j - 1$
- 10: **else**
- 11: substitute()
- 12: $i = i - 1$
- 13: $j = j - 1$
- 14: **end if**
- 15: **end while**

assign every kind of edit operation (insertion, deletion and substitution) the same edit cost, Algorithm 1 will give the left matrix in Figure 3.3 and an algorithm like Algorithm 2 allows us to find the *traces* in the left matrix for the shortest edit distance. A trace describes which symbols should be deleted, inserted or substituted by connecting neighbouring cells that have a minimum change in edit cost. When we place the edit operations that are needed when following a trace in a sequence, we obtain an edit transcript for the two sentences involved. For instance, there are three traces possible and thus three edit transcripts for S_1 and S_2 . These traces are depicted on the left side of Figure 3.2 and the edit transcripts are given in the left part of Figure 3.3. As explained in [Wagner and Fischer, 1974], changing

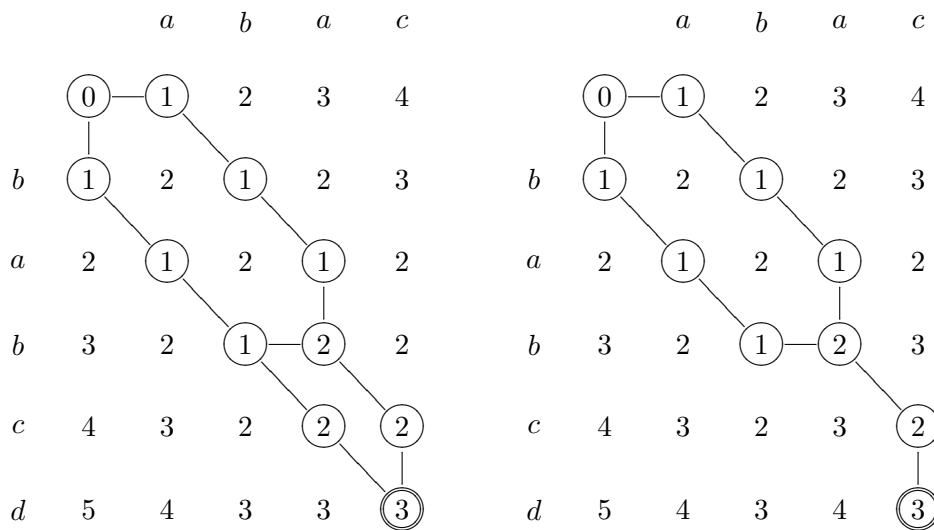


Figure 3.2: Traces when substitution cost is 1 (left) and 2 (right)

the edit cost of the edit operations would result in different traces and edit transcript. For instance, we could increase the substitution cost such that $c(E_d) = 1$, $c(E_i) = 1$ and $c(E_s) = 2$. When we do this for S_1 and S_2 , we will obtain the matrix as depicted on the right side of Figure 3.2 and the corresponding edit transcripts as given in the right part of Figure 3.3. Obviously, there are not many differences, but because substitution has become more costly, one of the traces has become more costly than the other two and is disqualified. As a result, the algorithm finds the *longest common substring*, which is valuable information in many applications and will be of use in the next chapter.

We can determine the complexity of the edit distance algorithm by considering that it takes $O(nm)$ to fill the $m \times n$ matrix using the recurrence relation. The traceback can be done in $O(n + m)$, considering that a trace in the matrix starts at one corner in the matrix, ends in the opposite corner of the matrix and is limited in direction. Thus, when we have p possible traces, they can be found in $O(p(n + m))$.

To reduce the complexity of dynamic programming algorithms such as the one we have described so far, there exists a method named *Four Russians*. In this method, the matrix

	E_d			E_s	E_s							
string 1	b	a	b	c	d							
string 2	-	a	b	a	c							
	E_d			E_i		E_d	E_d			E_i		E_d
string 1	b	a	b	-	c	d	b	a	b	-	c	d
string 2	-	a	b	a	c	-	-	a	b	a	c	-
	E_i			E_d		E_d	E_i			E_d		E_d
string 1	-	b	a	b	c	d	-	b	a	b	c	d
string 2	a	b	a	-	c	-	a	b	a	-	c	-

Figure 3.3: Edit transcripts for two strings when substitution cost is 1 (left) and 2 (right)

is divided into blocks of t cells and the values in the matrix are computed for each block rather than for each cell, resulting in a time complexity of $O(t)$ instead of $\theta(t^2)$. Using this method, the complexity of calculating the edit distance takes $O(nm/\log(m))$ [Masek and Paterson, 1980]. The only drawback in the use of the Four Russians method in aligning natural language sentences is that the algorithm is more effective on long strings whereas the natural language sentences are not long.

3.1.2 Using edit distance to locate substitutable subsentences

Whereas in the previous section we used letters as symbols of the alphabet, we now use words as symbols of the alphabet, like sentence 1³ and sentence 2⁴ in Figure 3.4. Note that

E	E_i		E_d	E_s		E_s	E_d
sentence 1	-	kijk	eens	in	de	grote	spiegel
sentence 2	ik	kijk	-	door	de	telescoop	-

Figure 3.4: Edit transcript for two sentences

when insertion and deletion have a cost of one and substitution has a cost of 2 that besides the edit transcript in Figure 3.4 there are three other edit transcripts possible that meet the minimal edit distance (= 7).

The substitutable subsentences are on those places where edit operations have been performed. Words in one sentence that did not undergo any edit operation with respect to the same location in the opposing sentence mark the beginning or end of a substitutable subsentence. In order to specify these subsentences in terms of indices in the context-sentences, we adopt and slightly modify the definition of *link* from [van Zaanen, 2002].

³Have a look in the big mirror.

⁴I am looking through the telescope.

Definition 3.3 (link). A link is a pair of indices $\langle i^S, j^T \rangle$ in two sentences S and T , such that $S[i^S] = T[j^T]$ and $S[i^S]$ is above $T[j^T]$ in the edit transcript of S and T .

For the example in Figure 3.4, the links are $\langle 1, 2 \rangle$ and $\langle 4, 4 \rangle$. In order to find all the links given the edit distance matrix, a slightly modified version of Algorithm 2 is needed. In case of two 'matching' alphabet symbols, we simply add a pair $\langle i, j \rangle$ to an, initially empty, set of pairs.

Given the links from an edit transcript, it is possible to derive the pairs of substitutable parts: $\langle -, S_2[1] \rangle$, $\langle S_1[2..3], S_2[3] \rangle$, $\langle S_1[5..6], S_2[5] \rangle$. How this can be done and how it can be used to hypothesize constituents is shown in Chapter 4.

3.2 Suffix trees

A suffix tree is a tree-like data structure for storing a string or, as we will see at the end of this chapter, for storing multiple strings. Suffix trees can be used to solve the exact string matching problem as introduced in the beginning of this chapter in linear time, achieving the same worst-case bound as the Knuth-Morris-Pratt and the Boyer-Moore algorithms. For a string S of length m , a substring T of length n can be matched in $O(n)$ time by first constructing a suffix tree for S . This preprocessing will take linear $O(m)$ time. The $O(m)$ preprocessing time and $O(n)$ search time makes suffix trees favorable with large strings or texts, since the search time depends on the length of T rather than on the length of S . As it will become clear in this chapter and the next, the data structure of the suffix tree will allow us to quickly find sentences that have equal parts and the places in these sentences where the equal parts are located.

In this chapter we start by describing and defining a suffix tree in the traditional context (i.e. applicable on strings) and we have a look to the space complexity and the construction time complexity. In Section 3.2.2 the algorithm to build a suffix tree is described. In the last section we consider the implications of constructing a suffix tree for a set of strings rather than on a single string.

3.2.1 Introducing suffix trees

3.2.1.1 Tries

Before we consider the properties of a suffix tree, we take a look at a similar data structure of which a suffix tree is derived: the *trie*. A *trie* is a rooted, labeled tree representing a set of strings. Each edge is labeled with a symbol and each node corresponds to the string that is a concatenation of the symbols on the path from the root to that node. The root represents

the empty string. A trie for the set T of strings where $T = \{abc, aba, bd, cca, ccb\}$ is depicted in Figure 3.5.

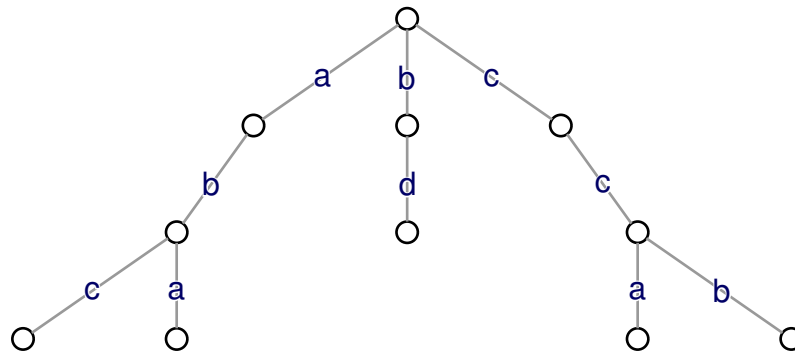


Figure 3.5: A trie for $T = \{abc, aba, bd, cca, ccb\}$

We can also use the trie data structure to represent all suffixes for a particular string. A *suffix trie* of a string S , denoted by $STrie(S)$, is a trie representing $Suffix(S)$, i.e. the set of all suffixes of string S .

From the description of a trie and from Figure 3.5 we can see that there are nodes that have one incoming edge and one outgoing edge. When we leave these nodes out of the tree, join each incoming and outgoing edge and subsequently concatenate the labels of the two edges in one label for the joined edge, we obtain a *compact trie*. We call the process of compacting a trie *path compression*. When we use a compact trie to represent the suffixes of a string we can call the resulting tree a *compact suffix trie*. An example of a non-compact suffix trie and a compact suffix trie is given in Figure 3.6.

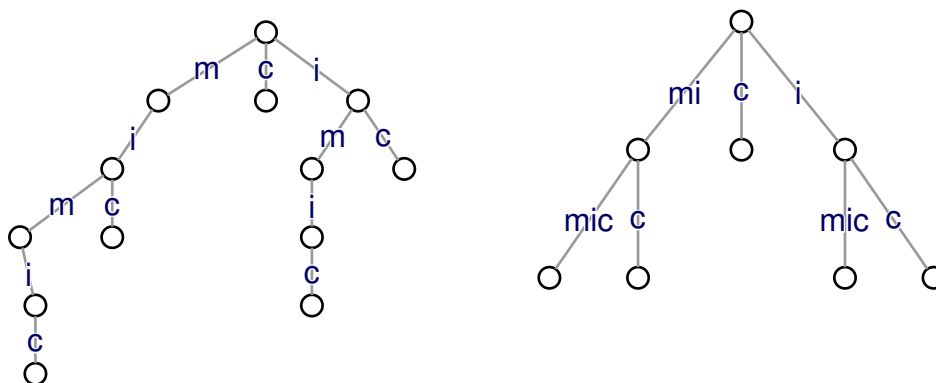


Figure 3.6: A non-compact *suffix trie* (left) and a compact *suffix trie* of $S = \text{mimic}$.

When we look to the space complexity of a suffix trie, either non-compact or compact, we can formulate the following theorem:

Theorem 3.2.1 (space for $STrie(S)$). The space required for $STrie(S)$ where $n = |S|$ is of complexity $O(n^2)$.

Proof. The size of $STrie(S)$ is linear in the number of substrings of S . S has at most $(n + 1)^2$ substrings. Thus, the size of $STrie(S)$ is $O(n^2)$. \square

The construction time of tries is strongly related to that of suffix trees and will not be treated separately.

3.2.1.2 Suffix Trees

We can define a suffix tree to be a compacted suffix-trie. An independent definition of a suffix tree is given below:

Definition 3.4 (suffix tree). A suffix tree T for a string S (with $n = |S|$) is a rooted, labeled tree with a leaf for each non-empty suffix of S . Furthermore, a suffix tree satisfies the following properties:

- Each internal node, other than the root, has at least two children;
- Each edge leaving a particular node is labeled with a non-empty substring of S of which the first symbol is unique among all first symbols of the edge labels of the edges leaving this particular node;
- For any leaf in the tree, the concatenation of the edge labels on the path from the root to this leaf exactly spells out a non-empty suffix of s .

For example, the suffix tree for the string $abac$ is shown in Figure 3.7.

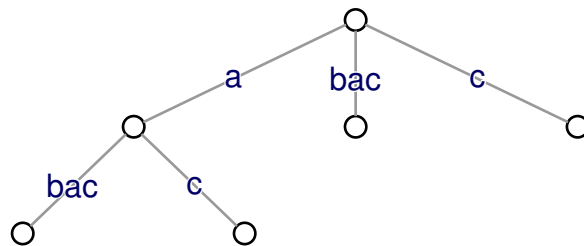


Figure 3.7: Suffix tree for $S = abac$.

When S ends with a unique character as depicted in Figure 3.7, then the suffix tree will have a leaf for each non-empty suffix and the tree conforms to Definition 3.4. When S does not end with a unique character, we will end up with a constructed tree like the third tree depicted in Figure 3.8 which represents the suffix tree of string aba . Here, the suffix a

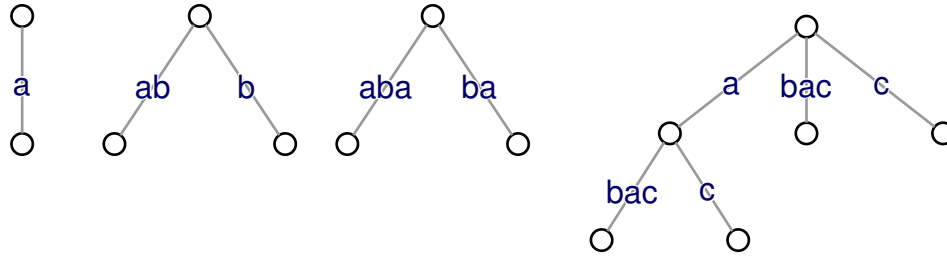


Figure 3.8: Implicit suffix trees for each suffix except ε in $S = abac$.

ends in an internal node. When S does not end with a unique character like in phase 3 of Figure 3.8, the resulting tree will have fewer leaves than suffixes because at least one of the suffixes of S is a prefix of another suffix. Henceforth, we call such a tree an *implicit suffix tree*. To refer to the positions of strings in a suffix tree that do not have a corresponding node in the suffix tree, due to path compression, we introduce the concept *point* in Definition 3.5.

Definition 3.5 (point). For each substring α of the string to represent in the suffix tree, $Point(\alpha)$ is a triple (v, d, c) where v is the node of maximum depth that represents a prefix of α , called β , where $d = |\alpha| - |\beta|$ and c is the $|\beta| + 1$ -st symbol of α when $\alpha \neq \beta$.

Less formally, we could say that when we traverse the suffix tree following the edges whose edge labels concatenate to the largest possible prefix of α , v is the last node on this path and d is the number of remaining symbols on the outgoing edge from v that starts with symbol c .

By adding a unique terminal symbol (e.g. $\$$) at the end of S , we can ensure that every non-empty suffix of S has a leaf again (Figure 3.8, phase 4). In conclusion, we can say that the final tree has n leaves, one for each non-empty suffix of $S\$$. Therefore, since each internal node has at least two outgoing edges and at maximum one incoming edge, the number of nodes is at most $2n$.

3.2.2 Ukkonen's suffix tree algorithm

There are three basic algorithms to build a suffix tree: Weiner's, McCreight's and Ukkonen's. Weiner [1973] presented the first linear time suffix tree construction algorithm. McCreight [1976] gave a simpler and less space consuming version, which became the standard in research on suffix trees. In this thesis, we consider Ukkonen's algorithm [1995] since it has the advantage over the algorithm of McCreight that it builds the tree incrementally from left to right instead of vice versa. This makes it more practical to use for applications such as data compression. Let us now take the same approach as in [Gusfield, 1997] by consider-

ing a naive way to construct a suffix tree and see how we can reduce the primary measures of complexity, running time and space usage, to $O(m)$ for both.

In the end, we want all m suffixes of string $S = a_1 \cdots a_m$ with $m = |S|$ to be present in the suffix tree. In order to do that we process each prefix ending at $S[i]$ of S , starting with $i = 1$ and incrementing i until $i = m$. For each prefix $S[1..i]$ we consider every suffix $S[j..i]$ where $1 \leq j \leq i$.

This will give us the following algorithm in pseudo-code:

Algorithm 3: Naive suffix tree construction

```

1: for  $i$  from 1 to  $m$  do { prefix phase }
2:   for  $j$  from 1 to  $i$  do { suffix phase }
3:      $\alpha = S[j..i - j]$ 
4:      $p = \text{Find\_end\_of\_path\_from\_root}(\alpha)$ 
5:      $\text{Apply\_suffix\_extension\_rules}(p, S[i])$ 
6:   end for
7: end for

```

3.2.2.1 Suffix extension rules

In Algorithm 3 the two for-loops already make time complexity of the algorithm $O(m^2)$. α represents the substring (suffix) that is not yet expressed by the suffix tree that has been built so far. The routine *Find_end_of_path_from_root()* finds the *point* in the tree for which the path from the root to this point is the maximum prefix that can be represented in the tree. Because in the worst case, this routine has to walk from the root node to a leaf node, it makes the naive algorithm of $O(m^3)$ complexity. The routine *Apply_suffix_extension_rules()* is of constant complexity as will become clear in the following three rules that are applied in the routine.

- *Rule 1* IF path α ends in a leaf node THEN $S[i]$ is added to the end of the label of that edge;
- *Rule 2* IF there is no path from the end of α continuing with $S[i]$ THEN:
 - *a.* IF α ends inside an edge THEN create a new internal node, splitting the edge in two edges.
 - *b.* Create a new leaf edge starting from α labeled with $S[i]$.
- *Rule 3* IF there is a path continuing with $S[i]$ where α ends THEN do nothing.

To illustrate these rules, let us consider the process of building a suffix tree for the string $S = abac$. We can see the process of constructing a suffix tree of S as creating and extending a

Table 3.1: Suffix extension rules to be applied when building up $STree(abc)$

1st tree	2nd tree	3rd tree	4th tree
$S[i] = a$	$S[i] = b$	$S[i] = a$	$S[i] = c$
$\alpha = \varepsilon \Rightarrow \text{R2b}$	$\alpha = a \Rightarrow \text{R1}$ $\alpha = \varepsilon \Rightarrow \text{R2b}$	$\alpha = ab \Rightarrow \text{R1}$ $\alpha = b \Rightarrow \text{R1}$ $\alpha = \varepsilon$	$\alpha = aba \Rightarrow \text{R1}$ $\alpha = ba \Rightarrow \text{R1}$ $\alpha = a \Rightarrow \text{R2a+b}$ $\alpha = \varepsilon \Rightarrow \text{R2b}$

suffix tree for each prefix in S . We start with just a single node: the root. Since we only have the root node we can immediately apply the suffix extension rules. From the root node there is no path continuing with $S[i] = a$ so we apply rule 2. Rule 2a is conditioned and does not apply in this case. 2b, however, always applies and we create a new leaf with edge starting from the root node and labeled with $S[i]$ (see the first tree of Figure 3.8). Likewise, the trees for the other prefixes are built and depicted in Figure 3.8. The corresponding values for α and $S[i]$ in each construction step as well as the rules applied can be found in Table 3.1.

3.2.2.2 Edge-label compression

In Definition 1.10 we defined an edge as a pair of nodes: $e_k = (v_i, v_j)$. In an edge-labeled tree we could define the label as a string L with $n = |L|$ and $n > 0$ and redefine an edge to be a triple consisting of two nodes and the label string: $e_k = (v_i, v_j, L)$.

When storing all the edge labels, the suffix tree requires $O(m^2)$ space: when we have a string S with $m = |S|$ there are $m + 1$ possible suffixes that can have a maximum length of m characters. We can reduce the space needed to store the edge labels by using a pair of pointers $\langle p, q \rangle$: $e_k = (v_i, v_j, \langle p, q \rangle)$. Here, a pointer consists of a numerical value that indicates a position in the input string. Thus, p points to the first symbol of the edge label in the input string whereas q points to the last symbol of the edge label in the input string.

Since $STree(S)$ has at most $2m - 1$ edges, the edge labels can occupy at most $2(2m - 1)$ space and thus we have reduced the necessary space for a suffix tree from $O(m^2)$ to $O(m)$. The edge-label compression is illustrated in Figure 3.9

3.2.2.3 Suffix links

In Algorithm 3, the routine *Find_end_of_path_from_root()* requires us for every suffix phase to traverse through the suffix tree and to consider the edge labels on the way until we find the path to the point where we can apply the suffix extension rules. It would be a saving in construction time when we could simply 'jump' from the end of the path of suffix phase $j - 1$ to the end of the path of the suffix phase j and avoid full traversal. This can be achieved by using *suffix links*.

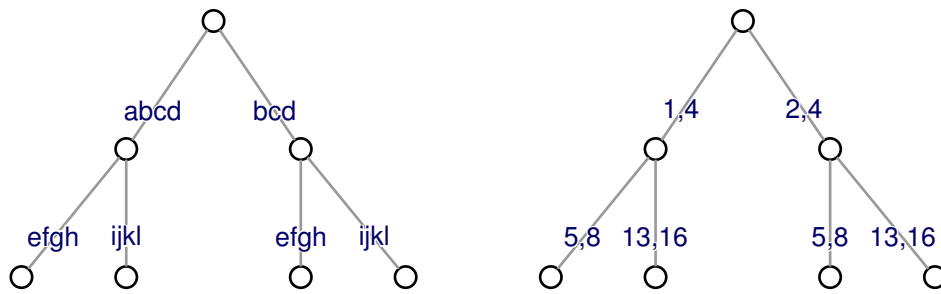


Figure 3.9: The left tree is a fragment of a suffix tree with explicit edge labels whereas the right tree shows pointers to positions in $S = abcdefghabcdijkl$.

Definition 3.6 (suffix link). Let S be a string and $STree(S) = (V, E)$ where V is a finite set of nodes: $V = \{v_1, v_2, \dots, v_n\}$ and E is a finite set of edges. Let $x\alpha$ be a string where $x \in \Sigma^+$ and α is a substring of S . Let a *path-label* of a node be the label of a path from the root to this node. Let v be a node ($v \in V$) with path-label $x\alpha$ and v is internal, if there is a $s(v) \in V$ with path-label α , then a pointer from v to $s(v)$ is called a *suffix link* and is denoted as the pair $\langle v, s(v) \rangle$. In the case when $x = \varepsilon$, the suffix link with path-label $x\alpha$ goes to the root node.

The suffix links are illustrated in Figure 3.10, where the suffix link $(1, 5)$ makes a jump from suffix bac to suffix ac possible.

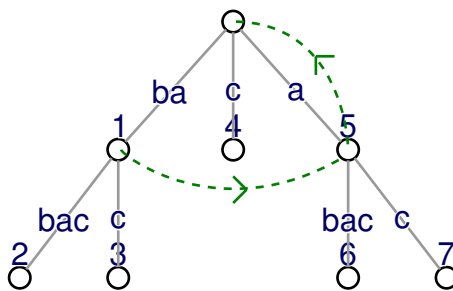


Figure 3.10: $STree(S)$ for $S = babac$ with suffix links $(1, 5)$ and $(5, root)$.

Using the suffix links, we only need to traverse from the root node to the leaf node in the first suffix phase of a particular prefix phase. The remaining suffixes in the prefix phase can be traversed by following the suffix links according to Algorithm 4.

Although the use of suffix links reduces the number of operations needed to find the end of a path in the suffix phase, we still cannot reduce $O(m^3)$ since the traversal of edges requires us to consider every full edge label. After following the suffix link, Algorithm 4

Algorithm 4: Less naive suffix tree construction (includes suffix links)

Require: S, α, γ : string

i, j, m : integer

p : point

```
1: for  $i$  from 1 to  $m$  do {prefix phase}
2:   for  $j$  from 1 to  $i$  do {suffix phase}
3:      $\alpha = S[j..i - j]$ 
4:     if  $j > 1$  then
5:       Find the first node  $v$  at or above the end of  $S[j - 1..i - 1]$  that either has a suffix
       link from it or is the root.
6:       Let  $\gamma$  (possibly empty) denote the string between  $v$  and the end of  $S[j - 1..i - 1]$ .
7:       if  $v \neq \text{root}$  then
8:         Traverse the suffix link from  $v$  to  $s(v)$ 
9:         Follow from  $s(v)$  the path  $\gamma$ 
10:      else
11:         $p = \text{Find\_end\_of\_path\_from\_root}(\alpha)$ 
12:      end if
13:    else
14:       $p = \text{Find\_end\_of\_path\_from\_root}(\alpha)$ 
15:    end if
16:    Apply\_suffix\_extension\_rules( $p, S[i]$ )
17:    if ( $j > 1$ ) and (an internal node  $w$  was created in suffix phase  $j - 1$ ) then
18:      Create a suffix link  $(w, v)$  in  $w$ 
19:    end if
20:  end for
21: end for
```

line 9 walks down from $s(v)$ following the path for γ . This walk takes time proportional to $|\gamma|$, the number of characters on the path to follow. We can reduce the time complexity to traverse an edge from its parent node to its child node to constant time by making use of the knowledge that no two edges leaving the same node can have a label beginning with the same character. Identifying and following an edge by only comparing the first character of the label makes the total time to traverse a path proportional to the number of nodes on the path, which is bound by $3m$ over a particular prefix phase⁵, making the total required time for traversing all the edges in a prefix phase $O(m)$.

When having reduced each suffix phase to $O(m)$, Algorithm 4 can be implemented to run in $O(m^2)$ time.

3.2.2.4 Reducing suffix phases to $O(1)$

When we can manage to bring down the running time of each suffix phase to constant time, our algorithm will be able to construct a complete suffix tree in $O(m)$ time. A formal proof goes beyond the scope of this thesis, but can be found in [Ukkonen, 1995]. In order to achieve this bound, we restrict the region in the tree we have to consider for possible updating according to the following principles.

Once a leaf node, always a leaf node

One of the characteristics of a suffix tree is that a node that is created to be a leaf will always remain a leaf node during the successive prefix phases. It will never be given a descendant and the label of the incoming edge can extend by suffix extension rule 1.

This means that we can extend all the edge labels of incoming edges to leaf nodes with $S[i]$ instead of suffix tree traversal and applying rule 1 for every suffix $S[j..i - j]$ of a particular prefix. Even more efficient is to set the edge labels of the incoming edges of leaf nodes to represent all the characters from the first character of the edge label to the last character of S . This will make rule 1 completely obsolete.

Confine the range for tree updating

In the previous paragraph, we concluded that leaf nodes will not contribute in changing the topology of the suffix tree. The introduction of the suffix link allows us to define a region in the suffix tree where the suffix tree can be changed and in conclusion restrict the updating and cut the running time complexity.

When updating a suffix tree, let us define the point of greatest depth where the suffix tree might need to be altered in prefix phase i the *active point* (AP for short) of the tree. All the suffixes that are longer than the suffix ending at the AP will end in leaf nodes whereas none of the suffixes that are shorter than the suffix ending at the AP will end in leaf nodes. Other points that might need to be altered can be found starting from the AP, traversing by suffix

⁵For proof, see [Gusfield, 1997]

links to the next smaller suffix ending at the smallest suffix: the empty string.

Using the AP opposed to the fixed root node as starting point of the updating process implies that we have to redefine the way how we indicate the ending point of a suffix in a suffix tree. Let us consider the implicit $STree(S)$ for $S = abacab$ in Figure 3.11. If the root would always be the starting point, we could simply refer to suffix ab as $suf = ab$. If the starting point is not fixed, we can specify the node that is the closest parent to then end of the suffix together with the remaining path from this node, referring to suffix ab as $suf = (3, b)$. The latter representation of the suffix is better known as the *canonical representation*.

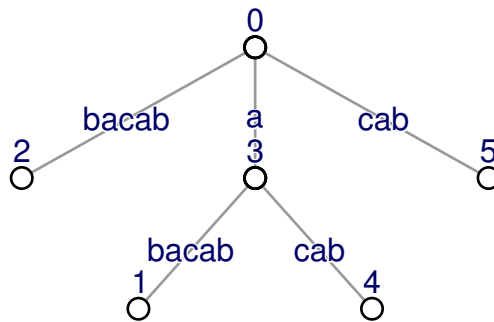


Figure 3.11: Implicit $STree(abacab)$.

In any prefix phase, if suffix extension rule 3 applies (i.e. the suffix ends in an edge) in suffix phase j , it will also apply in all further suffix phases ($j + 1$ to i) in the prefix phase. Thus, from the moment rule 3 applies for the first time in a prefix phase i , no alterations will happen in i and we can define this moment to be the *end point* (EP for short).

The AP and the EP define exactly the region in which the topology of the tree might change during updating. By restricting the update part of the algorithm to this region and combining this strategy with the knowledge gained in the previous paragraph: *once a leaf node, always a leaf node*, the suffix phase is restricted to $O(m)$ construction time complexity.

3.2.3 From strings to corpora

In the previous sections we have studied the construction of a suffix tree in the context of matching substrings in strings. However, we do not want to build a suffix tree for a particular string, but to build a suffix tree for a sentence. This involves a change in the alphabet: that of characters as symbols to words as symbols. Until now we have assumed that the alphabet is small and constant ($|\Sigma| = 26$), but when we consider a natural language the alphabet is considerably larger, depending on the nature of the corpus considered. In order to find an outgoing edge of a node in constant time, we would need an array of size $|\Sigma|$ for each node and subsequently have $O(m|\Sigma|)$ data structure size and time complexity. A practical way

to reduce complexity is to use a global hash table where the edges are stored and where the hash key consists of the concatenation of node identifier and the first symbol of the edge label. The time to find an edge can then be kept near-constant. The size of the hash table will have an upper bound of $2m$, since each internal node will have at most one parent.

In addition, we want to be able to build a suffix tree of an entire corpus, i.e. a collection of sentences. A corpus can, of course, be defined as a single string with a newline character or another character (e.g. full stop) as delimiter, but this would raise problems of both theoretical and practical point of view.

Whereas a suffix tree of a string has characters as elements, a suffix tree of a corpus has words as elements. Because characters take a low amount of space storage it has no sense to do something with recurring characters. However, when we consider the sentences in a corpus, words occupy a lot more space. A first optimization can be realized by representing every distinct word with a unique identifier (e.g. an integer value) and keeping a table with the words and their corresponding identifiers. In this way, recurring words have the same identifier and the space needed to represent the corpus can be reduced. This situation is depicted in Figure 3.12 for the subsentence *de man en de kat*⁶. When we have efficient

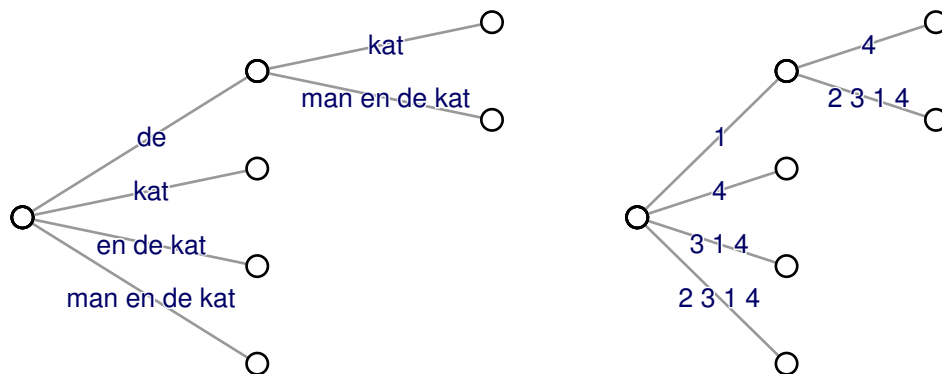


Figure 3.12: A suffix tree for $S = \text{de man en de kat}$ with word identifiers

storage of the words in the suffix tree we could, naively, consider the corpus as a single string with unique (alphanumerical) delimiters that mark the sentence. In Figure 3.13 the delimiters for sentence 1: *deman* and sentence 2: *dekat* are resp. '%' and '\$'. As can be seen in the figure, the problem with this approach is that without truncation, the suffixes for a sentence in the beginning of the corpus will continue till the end of the corpus. This problem can be solved by truncating every suffix just before the first sentence delimiter found. However, this solution is not elegant and the storage of a full size corpus in a single string may lead to implementational problems.

A better approach is to use a string for each single sentence. No explicit delimiters are

⁶*the man and the cat.*

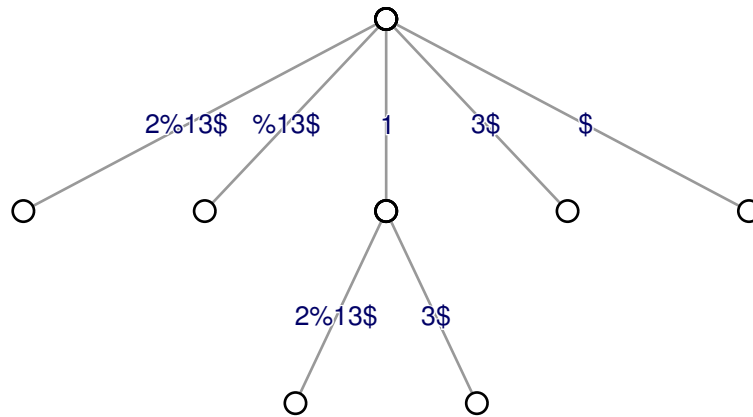


Figure 3.13: A suffix tree for *de man % de kat \$*

needed since the end of the string indicates the end of the sentence. This approach is more elegant and it requires only few changes to be made to the construction algorithm.

In the first place, we need to reconsider the way edge-label compression was defined in Subsection 3.2.2.2. Remember that in a particular edge, the pair of pointers $\langle p, q \rangle$ indicated the position of the first and last character of an edge label in the input string. But when the input consists of a set of sentences S , and we allow the pointers to be numerical values indicating a position in a string, we need to keep track of the sentence to which the pointers are referring to. For a particular edge k in the set of edges e , with $e_k = (v_i, v_j, \langle p, q \rangle)$, we should introduce a pointer r that points to a sentence in S , redefining an edge k as $e_k = (v_i, v_j, \langle p, q \rangle, r)$.

In the second place, we need to take a look to node- and edge creation. In general, we can assume that in the creation of a new edge while processing sentence s , r will refer to sentence s . In the *split operation*, we need to make sure that the edges involved are instantiated correctly and the edge label information is updated. Let us take a look to sentences S_1^7 and S_2^8 in Figure 3.14. Here, the left partial tree is constructed because of sentence 1. With processing sentence 2, two new edges will be introduced with $r = 2$ and the edge labels of the right partial tree will be constituted from more than once sentence. As mentioned earlier, the sentences are represented by ordered lists of word identifiers. The split operation involves addition of two new edges and a change in the label of the existent edge: the origin node of this edge changes and the start position p in the pair $\langle p, q \rangle$ of its label changes.

Now, it is possible to build a single suffix tree over a whole corpus of sentences. The

⁷the dog.

⁸the cat.

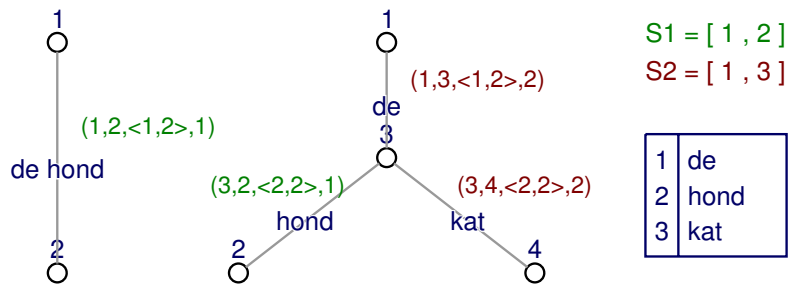


Figure 3.14: Split operation in joint suffix tree for $S_1 = de\ hond$, $S_2 = de\ kat$

maximum of sentences that can be represented by a suffix tree has no theoretical bound and is only bounded by computational resources.

Chapter 4

Learning by alignment

In the previous chapter we have introduced two approaches to compare sentences and find common substrings. In this chapter, we will consider how we can use both approaches to come to hypotheses for constituents. The first approach, by edit distance, has been pursued in [van Zaanen, 2002]. The second approach, by suffix tree, is brand new. In explaining the conditions on hypotheses and describing the hypothesis space we will follow the definitions and theorems as found in [van Zaanen, 2002]. The output of the algorithms, a set of hypotheses for constituents for each sentence, should have some properties that can be used to determine how successful the alignment learning was. This issue will be addressed in Chapter 5 where we will evaluate the output. As for the algorithms explained and constructed in this chapter, we aim to use the notion of substitutability to find as many regularities in natural language sentences as possible.

4.1 Hypotheses and the hypothesis space

In the first place, we need to determine whether a sentence that has changed because of substitution is still a correct sentence, i.e. is a *valid* sentence. We can do this by looking for occurrences of this sentence in the corpus and when at least one occurrence has been found, the sentence is considered valid.

Theorem 4.1.1 (validity). *A sentence S is valid iff an occurrence of S can be found in the corpus.*

Now we can define the notion of substitutability making use of validity as follows.

Definition 4.1 (substitutability). Let S and T be sentences with $n = |S|$ and $m = |T|$. Then, subsentences $u_{i...j}^S$ and $v_{k...l}^T$ are substitutable for each other if $w_{0...i}^S \cdot v_{k...l}^T \cdot w_{j...n}^S$ and $w_{0...k}^T \cdot v_{i...j}^S \cdot w_{l...m}^T$ are both valid.

From Definition 4.1 follows that sentence S can be transformed in sentence T by substituting one or more pairs of subsentences. When two subsentences can be substituted we can introduce two constituent hypotheses.

Theorem 4.1.2 (constituent hypotheses from subsentences). *If subsentences $u_{i\dots j}^S$ and $v_{k\dots l}^T$ are substitutable for each other then hypotheses $h_1 = \langle i, j, nt \rangle$ for sentence S and $h_2 = \langle k, l, nt \rangle$ for sentence T are introduced in the hypothesis space. Here, nt denotes the non-terminal that is common for both hypotheses.*

Because each sentence in the corpus is compared to other sentences, one sentence of the corpus can have many hypotheses. To store these hypotheses, each plain sentence is first converted to a structure called a *fuzzy tree*. This structure incorporates the sentence and keeps a list of one or more hypotheses for this sentence. We will depict such a fuzzy tree by the sentence involved and use labeled bracketing to indicate the hypotheses. Each of the fuzzy trees consists initially of the plain sentence and a hypothesis indicating that the sentence can be reached from the start symbol of the underlying grammar (i.e. the grammar to be learned). As such, we can depict the fuzzy trees of the two sentences *Jan loopt hard*¹ and *Jan loopt erg hard*² as in Example 4.1 and Example 4.2.

(4.1) [Jan loopt hard]₀

(4.2) [Jan loopt erg hard]₀

The algorithm that is needed to infer new hypotheses depends on the approach that is used. For both approaches, an algorithm is given and explained.

4.2 Using edit distance

In definition 3.3 we have defined a link. For the sake of clarity, we re-introduce it.

Definition 4.2 (link). A link is a pair of indices $\langle i^S, j^T \rangle$ in two sentences S and T , such that $S[i^S] = T[j^T]$ and $S[i^S]$ is above $T[j^T]$ in the edit transcript of S and T .

By creating links from an edit transcript of two sentences we can link the common words like in Figure 4.1. Here, the links are $\langle 1, 2 \rangle$, $\langle 4, 4 \rangle$ and $\langle 5, 5 \rangle$. When we combine adjacent links, such as the second and third link in Figure 4.1, we obtain equal subsentences. Two links $\langle i_1, j_1 \rangle$ and $\langle i_2, j_2 \rangle$ are adjacent when $|i_1 - i_2| = |j_1 - j_2| = 1$. From the maximal span of adjacent links we can construct a *word cluster*.

¹Jan walks fast

²Jan walks very fast

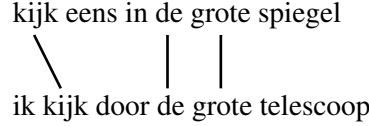


Figure 4.1: Links between common words of two sentences

Definition 4.3 (word cluster). A word cluster is a pair of subsentences $u_{i\dots j}^S$ and $v_{k\dots l}^T$ of the same length where $u_{i\dots j}^S = v_{k\dots l}^T$ and $i = k = 0$ or else $S[i - 1] \neq T[k - 1]$ and $j = l = |S|$ or else $S[j + 1] \neq T[l + 1]$.

Subsentences that are identified as word clusters describe those parts of sentences that are equal. To describe the parts of sentences that are unequal, we need to take the complement of the subsentences identified as a word clusters.

Definition 4.4 (complement). The complement of a list of subsentences

$[a_{i_1\dots i_2}^S, a_{i_3\dots i_4}^S, \dots, a_{i_{n-1}\dots i_n}^S]$ is the list of non-empty subsentences
 $[a_{i_0\dots i_1}^S, a_{i_2\dots i_3}^S, \dots, a_{i_n\dots i_{|S|}}^S]$

As soon as we have the indices of the complements, we have enough information to specify hypotheses. We can incorporate the approach with edit distance in an algorithm that infers new hypotheses by pairwise comparison of each fuzzy tree with each other fuzzy tree in the hypothesis space. After each comparison, the two fuzzy trees involved are enriched with newly inferred hypotheses and thus the hypothesis space is updated. This algorithm is given in Algorithm 5. Here, the procedure *FindSubstitutableSubsentences* finds the substitutable subsentences using edit distance when comparing two sentences whose fuzzy trees are given as arguments. The procedure *NewNonterminal* returns a new, unique non-terminal. This non-terminal is simply an unsigned integer that is incremented and returned each time the procedure is called. The procedure *AddHypothesis* adds its first argument, a hypothesis in the form $\langle b, e, n \rangle$, to the list of hypotheses of the fuzzy tree in its second argument.

The assumption that hypotheses in the same context will have the same non-terminal is not always correct. Consider, for instance, the following fuzzy trees as a result of alignment learning:

[the cat drinks [well]₁]₀
 [the cat drinks [milk]₁]₀

Although *well* is an adjective and *milk* is a noun, they are both assigned the same non-terminal because of their identical context. In this case, the introduction of such hypotheses

Algorithm 5: Edit distance alignment learning

Require: U : corpus
S: sentence
F,G: fuzzy tree
SS: list of pairs of pairs of indices in a sentence
PSS: pair of pairs of indices in a sentence
 B_F, E_F, B_G, E_G : indices in a sentence
N: non-terminal
D: hypothesis space

- 1: **for each** $S \in U$ **do**
- 2: $H := \{ \langle 0, |S|, \text{startsymbol} \rangle \}$
- 3: $F := \langle S, H \rangle$
- 4: **for each** $G \in D$ **do**
- 5: $SS := \text{FindSubstitutableSubsentences}(F, G)$
- 6: **for each** $PSS \in SS$ **do**
- 7: $\langle \langle B_F, E_F \rangle, \langle B_G, E_G \rangle \rangle := PSS$
- 8: $N := \text{NewNonterminal}()$
- 9: $\text{AddHypothesis}(\langle B_F, E_F, N \rangle, F)$
- 10: $\text{AddHypothesis}(\langle B_G, E_G, N \rangle, G)$
- 11: **end for**
- 12: **end for**
- 13: $D := D + F$
- 14: **end for**

is not a problem, because the merging of non-terminals belonging to the same constituent type can be done after the alignment on the basis of statistics.

4.3 Using suffix trees

If we want to use a suffix tree in a competitive alignment algorithm opposed to an alignment algorithm using edit distance that allows to learn from large corpora, we need to keep the time complexity of the competitive algorithm at most linear. As for the construction of a suffix tree, we have already seen that this can be done in $O(m)$, where m is the size of the corpus. The algorithm to derive the hypotheses from the suffix tree should be kept within linear bounds too.

Let us consider a mini-corpus U_1 consisting of two sentences $S_1, S_2 \in U_1$ as depicted in the left side of Figure 4.2 and see how we can derive hypotheses from its suffix tree. If we construct a suffix tree for U , its topology can be depicted as in Figure 4.2. Here, the numbers at the nodes of the tree are the node identifiers.

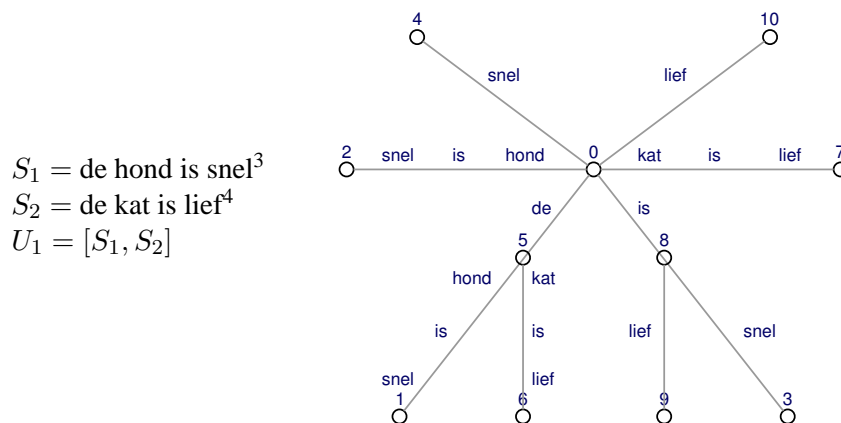


Figure 4.2: Corpus U_1 and $STree(U_1)$

Ideally, we would like to end up with two constituent types: one for the nouns and one for the adjectives. For the sake of simplicity, we ignore the start symbol of the underlying grammar that we discussed in Section 4.1.

$\text{de [hond]}_n \text{ is [snel]}_a$
 $\text{de [kat]}_n \text{ is [lief]}_a$

Because of the nature of the suffix tree, all words or word clusters distributed over the corpus that are alike have a joint edge starting at the root and ending at an internal node. In Figure 4.2 these edges are $\langle 0, 5 \rangle$ and $\langle 0, 8 \rangle$. The internal node marks the location in the sentences that have this node in their path where differences occur after having a word or word cluster in common. Thus, we can use the internal nodes to set the opening brackets of constituents:

$\text{de } [_n \text{ hond is } [_a \text{ snel}$
 $\text{de } [_n \text{ kat is } [_a \text{ lief}$

A first major advantage with respect to the edit distance approach is that we do not need to compare sentences pairwise. As soon as a suffix tree of the corpus is constructed, all edges leaving a particular internal node are supposed to indicate the start of the same constituent type in the sentences that use these edges.

To find the closing brackets of the constituents is somewhat more difficult. In the next subsections we first will look to the modifications in the suffix tree and the suffix tree con-

³the dog is quick.

⁴the cat is nice.

struction algorithm that are needed to indicate the position of the opening brackets of constituents. Then we will look to the possibilities of placing closing brackets. Our first attempt will be to place the closing brackets at the end of each sentence. Our second attempt will be to introduce the *prefix tree*, a suffix tree of the corpus with sentences with reversed word order, and use close brackets in the beginning and at the end of a sentence. The third approach will combine prefix- and suffix trees to capture infixes as well.

4.3.1 Where do hypotheses start

In Figure 3.14 (Subsection 3.2.3) we have seen how a split operation introduces an internal node in the suffix tree. In order to specify the edge labels, it was sufficient for each edge object to have a index variable that pointed to the sentence where the edge label could be identified. Thus, we redefined an edge k as $e_k = (v_i, v_j, \langle p, q \rangle, r)$ in which r points to a sentence in the corpus. Because of our desire for a particular edge e_k to know in which sentences and at which positions in these sentences e_k plays a role, we introduce a list L of sentence-index pairs and once more redefine an edge k as $e_k = (v_i, v_j, \langle p, q \rangle, r, L)$. The list L for each edge in a split operation is depicted in Figure 4.3. In the split operation, the list

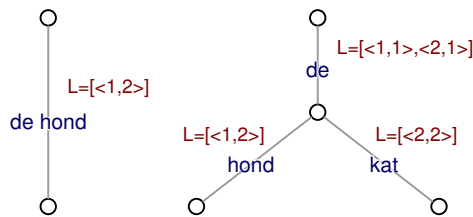


Figure 4.3: Sentence-position pairs before and after split operation

L of each newly introduced edge should be updated by combining the information of the edge in which the split occurs and the information of the newly introduced edge.

With the sentence information stored in each edge, we can construct an algorithm that considers each edge of the suffix tree and processes those edges that have more than one pair in L . Our only concern now seems to be the placement of the closing brackets of the constituents.

4.3.2 Closing hypotheses at the end of sentences

One simple solution to determine the position where to close the constituents is to place the closing brackets at the end of the sentence. In case of our example the bracketing results in:

$$\text{de } [{}_n \text{ hond is } [{}_a \text{ snel}]_a]_n$$

de [_n kat is [_a lief]_a]_n

The corresponding algorithm would be like Algorithm 6. The procedures *NewNonterminals* and *AddHypothesis* are the same as those used in Algorithm 5. In this algorithm, a suffix

Algorithm 6: Suffix tree alignment learning 1

Require: U : corpus
 S : sentence F : fuzzy tree
 L : list of sentence-index pairs
 I : index in a sentence
 N : non-terminal
 D : hypothesis space

- 1: **for each** $S \in U$ **do**
- 2: $H := \{ \langle 0, |S|, \text{startsymbol} \rangle \}$
- 3: $F := \langle S, H \rangle$
- 4: $D := D + F$
- 5: **end for**
- 6: Construct $STree(U)$
- 7: **for each** edge $k \in STree(U)$ leaving the root **do**
- 8: **if** $|L_k| > 0$ **then**
- 9: $N := \text{NewNonterminal}()$
- 10: **for each** pair $\langle S, I \rangle \in L_k$ **do**
- 11: $\text{AddHypothesis}(\langle I, |S|, N \rangle, F_S)$
- 12: **end for**
- 13: **end if**
- 14: **end for**

tree is build from the entire corpus. Each edge of the suffix tree is considered and those edges that are used in more sentences at the same time give rise to a new constituent type. A matter of concern in this algorithm based on suffix trees might be the fact that in case of a word cluster, a constituent type can be introduced multiple times depending on the span of the word cluster. For instance, let us take corpus U_2 and $STree(U_2)$ in Figure 4.4. Because *erg* is a suffix of *is erg* (there is a suffix link from node 6 pointing to node 8), the opening bracket after *erg* will be introduced twice in the two sentences. This will not be a problem because the updating of a fuzzy trees does not allow the presence of doubles and if the updating process would allow the presence of doubles they would be clustered together in a post-alignment cluster phase. Nevertheless, it is more elegant if a particular opening bracket is introduced only once. We can achieve this by making use of the possible presence of suffix links for the end node of the edge under consideration and determine if an edge is involved in representing a suffix of the edge label of an edge that has already been used to introduce one or multiple opening brackets.

⁵*he is very quick*

⁶*she is very nice*

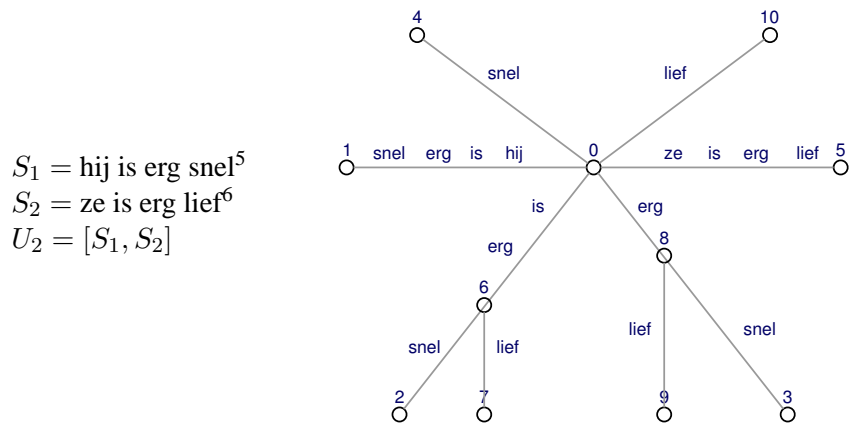


Figure 4.4: Corpus U_2 and $STree(U_2)$

As we have seen, a suffix tree allows us to find the positions in sentences where unequal parts initiate. Furthermore, we are also interested in the positions where unequal parts are followed by equal parts. We could use the same suffix tree for this purpose and consider the start positions of the edges from the root belonging to multiple sentences. However, in this situation the matter of concern described in the previous paragraph *does* pose a problem since the consideration of edge $\langle 0, 6 \rangle$ would correctly result in placing a closing bracket in front of *is erg* but the consideration of edge $\langle 0, 8 \rangle$ would incorrectly result in placing a closing bracket in front of *erg*. The use of a *prefix tree* solves this problem. A prefix tree is simply a suffix tree of the corpus with all sentences in reversed word order. When we construct a prefix tree for the corpus U_2 we get a suffix tree like depicted in Figure 4.5. Now we can use the prefix tree for placing closing brackets like we used the suffix tree for

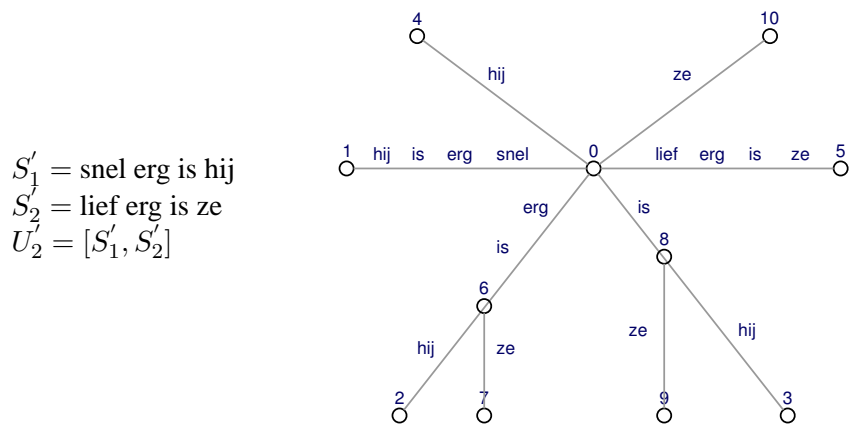


Figure 4.5: Corpus U'_2 and $STree(U'_2)$

placing opening brackets. We still have two joint edges, but now their labels end at the

position where differences start to occur. Now we can construct an algorithm that takes into account non-similar prefixes too (Algorithm 7) and place corresponding opening brackets at the beginning of sentences.

Algorithm 7: Suffix tree alignment learning 2

Require: U : corpus
 U' : reversed U
 S : sentence F : fuzzy tree
 L : list of sentence-index pairs
 I : index in a sentence
 N : non-terminal
 D : hypothesis space

- 1: **for each** $S \in U$ **do**
- 2: $H := \{ \langle 0, |S|, \text{startsymbol} \rangle \}$
- 3: $F := \langle S, H \rangle$
- 4: $D := D + F$
- 5: **end for**
- 6: Construct $\text{STree}(U)$
- 7: Construct $\text{STree}(U')$
- 8: **for each** edge $k \in \text{STree}(U)$ leaving the root **do**
- 9: **if** $|L_k| > 0$ **then**
- 10: $N := \text{NewNonterminal}()$
- 11: **for each** pair $\langle S, I \rangle \in L_k$ **do**
- 12: $\text{AddHypothesis}(\langle I, |S|, N \rangle, F_S)$
- 13: **end for**
- 14: **end if**
- 15: **end for**
- 16: **for each** edge $k \in \text{STree}(U')$ leaving the root **do**
- 17: **if** $|L_k| > 0$ **then**
- 18: $N := \text{NewNonterminal}()$
- 19: **for each** pair $\langle S, I \rangle \in L_k$ **do**
- 20: $\text{AddHypothesis}(\langle 0, I, N \rangle, F_S)$
- 21: **end for**
- 22: **end if**
- 23: **end for**

When we use the sentences from Figure 4.4, we would obtain the following bracketing:

$$[{}_2 \text{ hij}]_2 \text{ is erg } [{}_1 \text{ snel}]_1$$

$$[{}_2 \text{ ze}]_2 \text{ is erg } [{}_1 \text{ lief}]_1$$

This bracketing looks promising, but only has success when there is only one common part over the compared sentences. Preferably, we would like to match opening and closing brackets in such a way that they yield not only suffix- and prefix hypotheses, but also infix hypotheses. The only problem is that we do not know which opening bracket should be

matched with which closing bracket. Consider the following three sentences after placement of brackets according to their joint suffix tree and prefix tree:

ze [₁ loopt] ₂ snel] ₃ weg⁷

ze [₁ is] ₂ snel⁸

ze [₁ gaat] ₃ weg⁹

What we can do is pair each opening bracket at position p in a sentence with each following closing bracket in this sentence that occurs after p and keep a storage with global scope where these pairs are assigned a joint identifier. In the sentences on the left side of Figure 4.6, the opening and closing brackets are placed according to information from the suffix tree and prefix tree respectively. The table in the middle of the figure represents the storage that contains the joint identifier for each unique pair of brackets. The sentences on the right side of the figure are those where the storage is used to replace the identifier of the brackets in one pair with the joint identifier. For the sake of simplicity, we omit placing an initial opening and a closing bracket at the beginning and the end of each sentence.

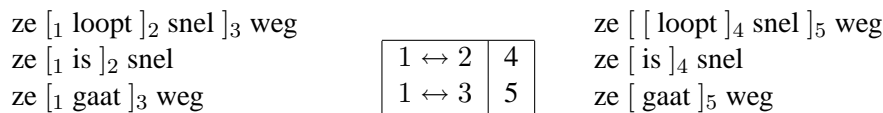


Figure 4.6: Matching opening and closing brackets in hypotheses generation.

A major difference with earlier algorithms is that we can now distinguish two phases. In the first phase, we use a data object P to keep for each sentence two sets of position elements: one for the opening brackets and one for the closing brackets. The set corresponding to the opening brackets is populated using information from the suffix tree. This set for a sentence S is denoted as $suffixset(S)$. The set corresponding to the closing brackets is populated using information from the prefix tree. This set for a sentence S is denoted as $prefixset(S)$. In the second phase, for each sentence S , brackets from the suffix set are matched with brackets from the prefix set and the resulting hypotheses are added to the fuzzy tree of sentence S . An algorithm that makes this possible is Algorithm 8.

In this algorithm, the procedure *AddBracket* adds a bracket position to P . The procedure takes as arguments the type of bracket (opening/closing), the sentence and the position in this sentence where the bracket should occur.

In conclusion of this chapter, we have introduced one alignment algorithm based on edit distance and three alignment algorithms based on the suffix tree data structure. The first algorithm (Algorithm 5) uses edit distance computation to introduce hypotheses. This algorithm has already been deployed in ABL. The second algorithm (Algorithm 6) uses the

⁷she walks away quickly

⁸she is quick

⁹she leaves

Algorithm 8: Suffix tree alignment learning 3

Require: U : corpus
 U' : reversed U
 S : sentence F : fuzzy tree
 P : data object for bracket positions
 L : list of sentence-index pairs in suffix tree
 I : index in a sentence
 N : non-terminal
 T : type of bracket (opening/closing)

- 1: Construct $STree(U)$
- 2: Construct $STree(U')$
 { Phase 1 }
- 3: **for each** edge $k \in STree(U)$ leaving the root **do**
- 4: **if** $|L_k| > 0$ **then**
- 5: **for each** pair $\langle S, I \rangle \in L_k$ **do**
- 6: AddBracket(T, S, I)
- 7: **end for**
- 8: **end if**
- 9: **end for**
- 10: **for each** edge $k \in STree(U')$ leaving the root **do**
- 11: **if** $|L_k| > 0$ **then**
- 12: $N := \text{NewNonterminal}()$
- 13: **for each** pair $\langle S, I \rangle \in L_k$ **do**
- 14: AddBracket(T, S, I)
- 15: **end for**
- 16: **end if**
- 17: **end for**
 { Phase 2 }
- 18: **for each** sentence $S \in U$ **do**
- 19: **for each** $I_1 \in \text{suffixset}(S)$ **do**
- 20: **for each** $I_2 \in \text{prefixset}(S)$ where $I_2 > I_1$ **do**
- 21: $N := \text{NewNonterminal}()$
- 22: AddHypothesis($\langle I_1, I_2, N \rangle, F_S$)
- 23: **end for**
- 24: **end for**
- 25: **end for**

suffix tree to determine the start position of a hypothesis and closed the hypothesis at the end of the sentence. The third algorithm (Algorithm 7) introduces additional hypotheses of which the stop position is determined by information from the prefix tree and the start position is the beginning of the sentence. The last algorithm (Algorithm 8) matches start positions derived from the suffix tree with stop positions derived from the prefix tree. When we look to the three suffix tree algorithms, it is evident that their output is different. The first suffix tree algorithm introduces the lowest number of hypotheses whereas the last suffix tree algorithm introduces the highest number of hypotheses and produces output that is the most equivalent to the output of the edit distance algorithm.

Chapter 5

Empirical Results

In this chapter, we will evaluate how both approaches, alignment using edit distance and alignment using suffix trees, perform with respect to a full learning task of grammatical inference. First, the framework in which both approaches will be evaluated (ABL) is described. Second, the metrics used to evaluate the output of the algorithms are explained. Third, the results of both approaches on several corpora of various size are presented and discussed.

5.1 ABL framework

The environment in which both approaches of alignment are tested is that of ABL. This framework can be divided in several phases¹, depending on the parameters of the system. Here, we divide the framework in the *alignment learning* phase and the *selection learning* phase.

In the *alignment learning* phase, possible constituents (hypotheses) are found in plain text sentences using a particular alignment method. Thus, the input for this phase is a plain text corpus and the output of this phase consists of each plain sentence followed by the possible constituents found for each sentence. In Example 5.1, the constituent hypotheses are again in the format (*start position, stop position, [non-terminal]*).

```
(5.1) INPUT   she is nice
        he is nice too
        OUTPUT she is nice (0,3,[0])(0,1,[1])(3,3,[2])
        he is nice too (0,4,[0])(0,1,[1])(3,4,[2])
```

By default, the sentences are aligned and hypotheses proposed using the method of computing the edit distance for each pair of sentences. Since we now have a new approach based on

¹A more elaborate description of each phase and possible instantiations of phases can be found in [van Zaanen, 2002].

suffix trees, this approach can be used as an alternative to do the alignment learning phase. In [van Zaanen, 2002], two variants of alignment learning using edit distance are introduced which are called *default* and *biased*. The difference is in the cost for the edit operations. Where the *default* variant simply uses the costs 1, 1, 2 for insertion, deletion and substitution respectively, the *biased* variant uses the costs 1, 1 for insertion and deletion too, but uses a cost function for substitution that favors substitution of parts that are relatively closer to each other².

After alignment learning, two or more constituent hypotheses in a sentence can overlap. This will be the case for the second sentence in Example 5.2. The hypotheses are shown using bracketing where the label of the bracket denotes its non-terminal.

(5.2) $[_1$ He buys $]_1$ a book
 $[_1$ She $]_2$ reads $]_1$ a book $]_2$
 She $]_2$ walks often $]_2$

Now, if we assume that the grammar to be learned is context-free and the bracketing reflects the rewriting of non-terminals in a derivation of the sentence, no overlapping constituents should be present and a choice must be made for one of the hypotheses. These choices are being made in the *selection learning* phase. In this phase, the best hypotheses are selected. This can be done in a non-probabilistic and a probabilistic way. In case of the latter, the probability of each hypothesis is computed and the hypotheses with higher probability are preferred.

In describing the alignment learning phase, we have implied that this phase can be done by both the edit distance and suffix tree approaches. However, because the edit distance approach compares sentences pairwise, hypotheses that occur in the same context do not all receive the same non-terminal and, subsequently, we need to do some post-alignment processing before initiating the selection learning phase. For instance, consider the result of the alignment learning phase using edit distance:

(5.3) INPUT she walks.
 she sleeps.
 she eats.
 OUTPUT she walks (0,2,[0])(1,2,[1,2])
 she sleeps (1,2,[1,3])(0,2,[0])
 she eats (1,2,[2,3])(0,2,[0])

By the knowledge that non-terminals 1 and 2 are identical and the knowledge that non-terminals 2 and 3 are identical, we can infer that 1, 2, 3 are identical and replace them with just one non-terminal.

²for more details on the biased variant, see [van Zaanen, 2002]

The performance of the overall system depends on the performance of the alignment learning phase (in providing as many correct hypotheses as possible and at the same time as few incorrect hypotheses as possible) and the selection learning phase (in selecting the right hypotheses as the correct constituents).

5.2 Evaluating performance

To evaluate the performance of a language learning system, several methods can be used. The method used here requires the existence of a *treebank*: a corpus where each of the sentences is bracketed such that the bracketing indicates the parse tree(s) of this sentence. An example of a sentence from a treebank with the corresponding visualized tree is given in Figure 5.1. In the method we are using, we consider the syntactic structures in the treebank

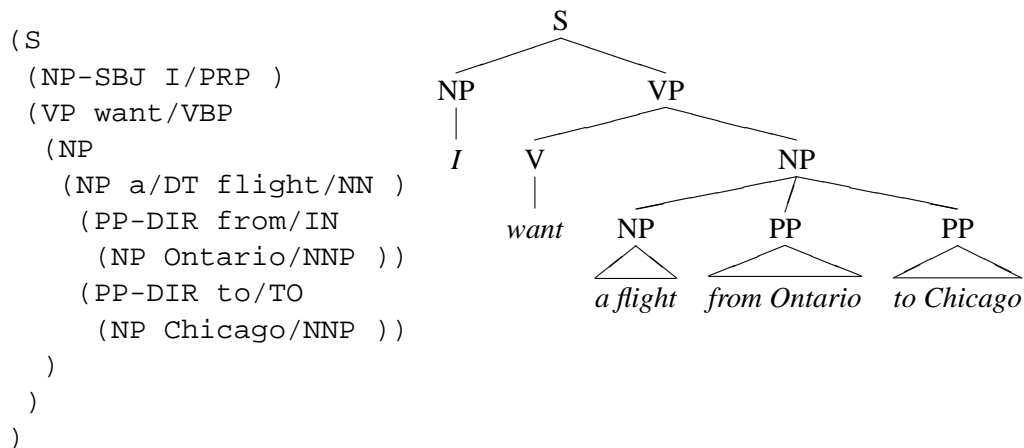


Figure 5.1: Treebank fragment for the sentence *I want a flight from Ontario to Chicago*

a gold standard (we consider it to be completely correct). From this ‘original’ treebank, a corpus of plain sentences is extracted and given as input for the learning process. The results of the learning process are stored in another treebank. By comparing the learned treebank with the original treebank, the learning process can be evaluated. We can do this by counting the number of constituents in the learned treebank that also occur in the gold standard treebank and use metrics such as recall and precision to determine the performance of the grammar that has been learned. The evaluation method described is depicted in Figure 5.2.

Before defining the metrics precision and recall in the context of constituents, we define S to be the set of all sentences. For sentence $s \in S$, the function $o(s)$ returns the tree of s from the original treebank and the function $l(s)$ returns the tree of s from the learned treebank. The function $c(t, u)$ returns the set of constituents that both occur in tree t and tree u . By taking the cardinality of a tree, we expect to get the number of constituents in this tree in

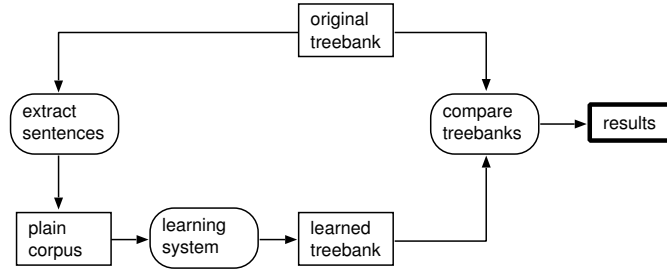


Figure 5.2: Evaluating a learned treebank

return. In order to express the performance of the learning system, we use three metrics well known in information retrieval. The first one, *recall*, shows the percentage of constituents from the original treebank that have been learned:

Definition 5.1 (constituent recall).

$$\text{recall} = \sum_{s \in S} \frac{|c(o(s), l(s))|}{|o(s)|}$$

The second one, *precision*, shows the percentage of constituents found that are correct:

Definition 5.2 (constituent precision).

$$\text{precision} = \sum_{s \in S} \frac{|c(o(s), l(s))|}{|l(s)|}$$

Occasionally, it is desirable to express both recall and precision in one metric: the *F-score*. This metric has a parameter, β , that makes the precision more important when being increased. Usually, $\beta = 1$, which makes recall and precision both as important.

Definition 5.3 (F-score).

$$F(\beta) = \frac{(\beta^2 + 1) \times \text{precision} \times \text{recall}}{(\beta^2 \times \text{precision}) + \text{recall}}$$

As might be clear from Section 5.1, we are particularly interested in how the alignment learning phase performs when using Levenshtein distance and when using suffix trees. In order to compare the performance on the learning task, we have to keep the performance of the selection learning phase constant. We do this by assuming the perfect selection learning. This means that from the hypotheses in its input, the selection learning process selects only those hypotheses that are also present in the original treebank. Since only correct hypotheses are selected from the learned treebank, the precision in these kind of experiments

is 100% and the recall indicates how many of the correct constituents are selected from the hypothesis space. The selection learning can never improve on these metric values. When we incorporate the two main phases of ABL in the evaluation scheme of Figure 5.2, and simulate the perfect selection learning with a process we call *max_score*, we can depict the evaluation environment as in Figure 5.3.

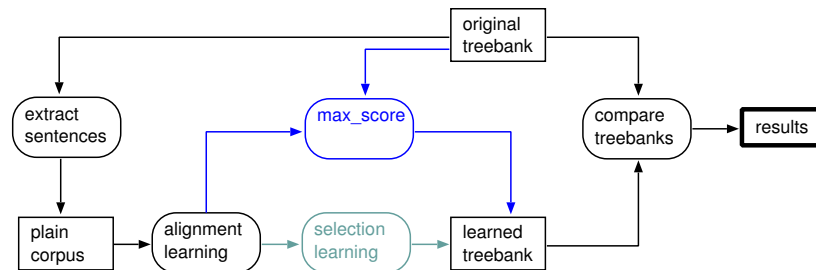


Figure 5.3: Evaluation with ABL

5.3 Performance on the ATIS corpus

The ATIS (Air Traffic Information System) treebank is taken from the Penn treebank [Marcus et al., 1993] and contains mostly questions and imperatives on air traffic. It contains 568 sentences with a mean length of 7.5 words per sentence and the lexicon size is 358 words. Examples of sentences from ATIS are given in Example 5.4.

- (5.4) How much would the coach fare cost
 Show me the flights from Baltimore to Seattle
 Which are nonstop

The results of applying the alignment learning phase to the ATIS corpus and selecting only the hypotheses that are present in the original treebank is given in Table 5.1. Here we use the results for the edit distance alignment learning as presented in [van Zaanen, 2002]³ after having reproduced them and verified their validity. Besides the *default* and *biased* variants of alignment learning with edit distance, we also use a baseline, which is called *random* because it randomly chooses between the two variants already mentioned and an additional variant which aims to introduce almost all alignments and leaves a huge load on the selection learning phase. As for the suffix tree alignment methods, we consider the three methods we introduced in Chapter 4; we abbreviate Algorithm 6 as *suf*, Algorithm 7 as *presuf 1* and Algorithm 8 as *presuf 2*.

³Where van Zaanen uses a slightly different ATIS version of 577 sentences, we use the original Penn treebank 2 ATIS version of 568 sentences.

Table 5.2 gives an overview of the number of hypotheses in the hypothesis space at two moments in the learning process. In the first place we count the number of hypotheses as the result of the alignment learning phase (column *learned*). In the second place, we count the number of hypotheses as the result of the selection learning phase (column *best*). By considering the two counts we can see how ‘generative’ the alignment learning was and how many hypotheses were left in the hypothesis space (column *% left*). To get an idea how the counts relate to the number of constituents in the original treebank, we counted 7,017 constituents for the ATIS. Note that the performance of the edit distance methods depends on the order of the sentences in the corpus whereas the performance of the suffix tree methods is invariant (hence their zero standard deviation).

Table 5.1: Results for alignment learning on the ATIS corpus

		recall		precision		F-score	
edit distance	random	28.90	(0.85)	100.00	(0.00)	44.83	(0.70)
	default	48.08	(0.09)	100.00	(0.00)	64.94	(0.08)
	biased	19.52	(2.67)	100.00	(0.00)	32.60	(3.64)
suffix tree	suf	27.41	(0.00)	100.00	(0.00)	43.03	(0.00)
	presuf 1	31.25	(0.00)	100.00	(0.00)	47.62	(0.00)
	presuf 2	38.35	(0.00)	100.00	(0.00)	55.44	(0.00)

Table 5.2: Number of hypotheses after alignment learning on the ATIS corpus

		learned		best		% left
edit distance	random	4,353	(0.0)	1,851	(25.6)	42.5
	default	12,692	(8.8)	4,457	(4.4)	35.1
	biased	2,189	(796.8)	1,175	(331.2)	53.7
suffix tree	suf	2,966	(0.0)	1,680	(0.0)	56.6
	presuf 1	5,780	(0.0)	2,223	(0.0)	38.5
	presuf 2	13,460	(0.0)	4,204	(0.0)	31.2

When comparing the results of the three suffix tree alignment methods, we can indeed observe the differences in the number of learned hypotheses as expected. We see a bigger difference between method *suf* and *presuf 1* than difference between *presuf 1* and *presuf 2*. This can be explained by considering the right-branching nature of the English language. Since prefix hypotheses implicitly have a left-branching nature, the effect of including prefix hypotheses (*presuf 1*) will not result in a high increase in recall. In the same line of thought, we can predict the method *suf* to perform worse on corpora of left-branching languages such as Japanese.

When we compare the recall of suffix tree alignment learning with edit distance alignment learning we can see that all suffix tree methods outperform method *biased* and *presuf 1* and *presuf 2* perform better than our baseline method (*random*). Although *presuf 2* performs reasonably well, the default method performs better significantly. As for the number of hypotheses that are found, we can observe that there are no major differences. The methods *default* and *presuf 2* both introduce the most hypotheses.

5.4 Performance on the OVIS corpus

The OVIS (Openbaar Vervoer Informatie Systeem)⁴ is a Dutch treebank that contains mostly questions, imperatives and answers in on public transport in the Netherlands. It contains 10,000 trees of sentences with a mean length of 3.5 words. Examples of sentences from OVIS are given in Example 5.5.

- (5.5) Ik wil graag van Den Haag Mariahoeve naar Amsterdam Centraal.⁵
 Nee dank u.⁶
 om half twaalf.⁷

Because the OVIS contains many trees of one-word ‘sentences’, in which we are not interested, we will use a subset of the OVIS in which one-word entries are removed. This subset has 6,797 sentences with a mean sentence length of 4.6 words. The lexicon of this subset contains 824 words and the subset itself has 54,452 constituents.

The results of applying the various alignment learning methods are given in Table 5.3 and the numbers of constituents are given in Table 5.4.

Table 5.3: Results for alignment learning on the OVIS corpus

		recall		precision		F-score	
edit distance	random	52.73	(0.09)	100.00	(0.00)	69.05	(0.40)
	default	94.22	(0.04)	100.00	(0.00)	97.02	(0.02)
	biased	53.65	(2.27)	100.00	(0.00)	69.81	(1.93)
suffix tree	suf	32.68	(0.00)	100.00	(0.00)	49.26	(0.00)
	presuf 1	51.31	(0.00)	100.00	(0.00)	67.82	(0.00)
	presuf 2	59.18	(0.00)	100.00	(0.00)	74.46	(0.00)

Table 5.4: Number of hypotheses after alignment learning on the OVIS corpus

		learned		best		% left
edit distance	random	34,221	(0.0)	22,301	(108.1)	56.2
	default	123,699	(62.6)	50,365	(28.7)	40.7
	biased	40,399	(1,506.6)	21,488	(1,049.8)	53.2
suffix tree	suf	20,065	(0.0)	11,005	(0.0)	54.8
	presuf 1	31,536	(0.0)	18,758	(0.0)	59.5
	presuf 2	51,890	(0.0)	27,007	(0.0)	52.0

When we compare the recall of the best edit distance method (default) with that of the best suffix tree method (presuf 2), we can see that method default performs very well with 94.22 percent recall. Although presuf 2 performs better than the baseline and the method biased,

⁴Literally translated as *Public Transport Information System*.

⁵*I want to go from Den Haag Mariahoeve to Amsterdam Central Station.*

⁶*No thank you.*

⁷*at half past eleven.*

the difference with method presuf 2 is remarkable. A first look on a small subset of the data (say ~40 sentences) produced by both methods does not show much differences in the hypotheses proposed and applying both methods on this subset results in the same recall values. However, on bigger subsets the both recall values start to diverge. When we look at the number of hypotheses, it becomes clear that at the end, the default method introduces more hypotheses (at least a factor 2) than the presuf 2 method.

5.5 Performance on the Wall Street Journal corpus

The WSJ (Wall Street Journal) corpus consists of newspaper articles and is more complex than the questions and imperatives in the ATIS and OVIS corpora. The WSJ has also a much larger lexicon. It consists of several sections of which we shall use section 23, which contains 1,904 sentences with an average sentence length of > 20 words and has informally developed as test section of the WSJ corpus⁸. The lexicon of section 23 contains 8,135 words and the section has 65,382 constituents. Examples of sentences from section 23 of the WSJ corpus are given in Example 5.6.

(5.6) Big investment banks refused to step up to the plate to support the beleaguered floor traders by buying big blocks of stock.

Once again the specialists were not able to handle the imbalances on the floor of the New York Stock Exchange.

When the dollar is in a free-fall, even central banks can't stop it.

For the WSJ, we choose to test only with the best edit distance alignment method (default) and the best suffix tree alignment method (presuf 2). When we apply these algorithms to section 23 of the WSJ we obtain the results as presented in Table 5.5. The numbers of constituents are given in Table 5.6.

Table 5.5: Results for alignment learning on section 23 of WSJ

		recall	precision	F-score
edit distance	default	53.26 (0.07)	100.00 (0.00)	69.50 (0.12)
suffix tree	presuf 2	52.05 (0.00)	100.00 (0.00)	68.46 (0.00)

Table 5.6: Number of hypotheses after alignment learning on section 23 of WSJ

		learned	best	% left
edit distance	default	186,593 (94.3)	34,835 (19.7)	18.7
suffix tree	presuf 2	280,789 (0.0)	34,031 (0.0)	12.1

In addition, we are interested what the results are for alignment learning on a large part of the WSJ corpus. For this purpose, we use sections 02-21 (standard set) and 23 (test set) together.

⁸See e.g. [Charniak, 1997; Collins, 1997; van Zaanen, 2002].

With a corpus of this size (38,009 sentences), edit distance alignment learning is not feasible anymore. For this reason, we present the results for the suffix tree alignment learning only (Table 5.7 and Table 5.8). The lexicon contains 40,982 words and this treebank fragment has 1,355,210 constituents.

Table 5.7: Results for alignment learning on WSJ section 2-21+23

		recall	precision	F-score
suffix tree	presuf 2	37.27 (0.00)	100.00 (0.00)	54.30 (0.00)

Table 5.8: Number of hypotheses after alignment learning on WSJ section 2-21+23

		learned	best	% left
suffix tree	presuf 2	3,995,083 (0.0)	541,162 (0.0)	13.7

As can be observed, the recall has dropped considerably and more than 80 percent of the constituents that have been introduced are removed. Compared with the performance on the previous treebanks, these results are bad. It might be the case that certain types of hypotheses that are valid constituents are not found and that this fraction of the total valid constituents to be found gets larger with the corpus size.

5.6 Performance on execution time

Another type of comparison between the different approaches of alignment learning is to consider the time it takes for both approaches to align a certain corpus. In Section 3.1, we determined the complexity of comparing two strings of n and m length using edit distance to be at least $O(nm) + O(p(n + m))$, where p was the number of traces possible. When we consider both n and m to be constant (the maximum sentence length), the complexity of aligning two corpus strings is linear with the number of traces for these two sentences. However, as soon as we start to compare each sentence with each other as described in Algorithm 5 (Chapter 4) we introduce a quadratic complexity in the number of sentences of the corpus. As for suffix trees, there is no need of pairwise comparing sentences. Thus, the complexity will be linear in the number of sentences of the corpus. This theoretical difference can be recognized in the figure we obtain when we plot the execution time required for aligning using both approaches as a function of the number of sentences in the corpus. This comparison is important in the sense that fast alignment learning would allow ABL to learn on large corpora as well. When we test both approaches with the first 6,000 sentences of the OVIS corpus, we obtain the timing curves depicted in Figure 5.4.

Note that the execution time in seconds depends on distributional characteristics of the corpus used. Also the performance of the workstation used in the alignment learning⁹ is

⁹The workstation used for all timing experiments is equipped with a 32-bit CPU clocking 2,495 MFLOPS and 1.0 GB of RAM.

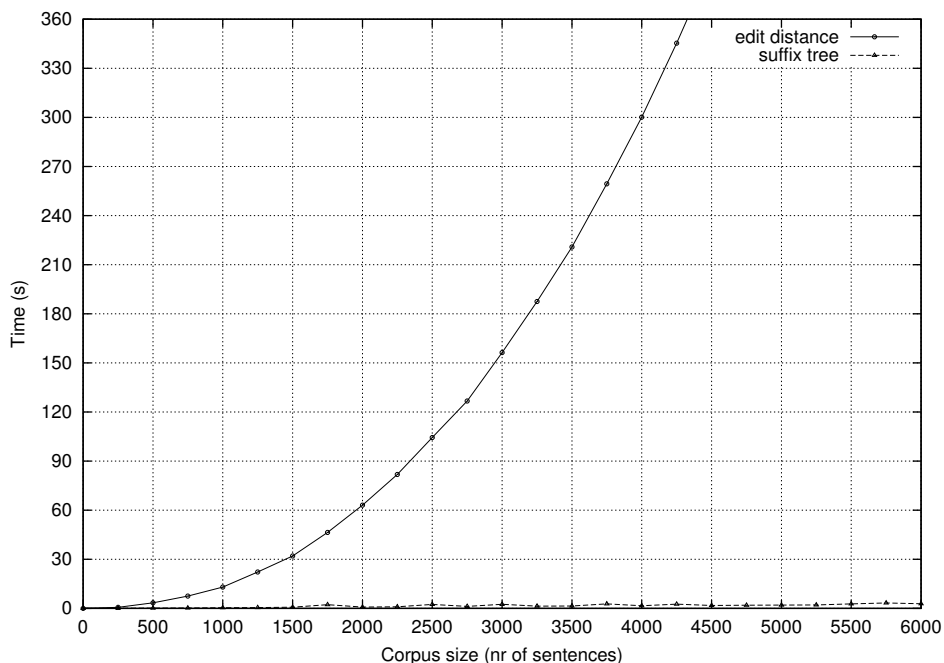


Figure 5.4: Execution time for edit distance and suffix tree methods

a reason not to focus on the units, but to focus on the shape of the curves in which linear progression and quadratic progression can clearly be observed.

When we look to the results of timing the alignment learning on the corpora we have used (Table 5.9), we can observe the merits of aligning by suffix trees.

Table 5.9: Timing results (in seconds) for edit distance alignment and suffix tree alignment on several corpora

	edit distance		suffix tree	
ATIS	4.05	(0.04)	0.37	(0.03)
OVIS	1107.63	(99.91)	2.87	(0.10)
WSJ 23	283.00	(0.73)	6.08	(0.39)
WSJ 2-21+23	n.a.	n.a.	124.86	(0.11)

On the ATIS corpus, suffix tree alignment is approximately a factor ~ 11 faster than edit distance alignment. On the OVIS corpus, suffix tree alignment is even a factor ~ 386 faster. On WSJ section 02-21+23, suffix tree alignment needed about 2 minutes whereas the edit distance alignment learning was aborted after almost 13 hours of computation.

Chapter 6

Conclusions and Future work

6.1 Conclusions

As presented in Chapter 2 there are several approaches for unsupervised grammatical inference. We have described approaches that use likelihood with Probabilistic Context-Free Grammars, Minimum Description Length and distributional information.

From this last kind of approach, we focus on Alignment-Based Learning (ABL) and study the computation of edit distance in order to align sentences and propose hypothesis constituents for substitutable subsentences. We conclude that despite possibilities to speed up the computation of the edit distance and the edit transcripts, edit distance alignment is of quadratic complexity in the number of corpus sentences.

Subsequently, we introduce the suffix tree data structure and the algorithm to build it. We describe how to make the construction time of the suffix tree of linear complexity. To build a suffix tree representing all the sentences in a corpus, the suffix tree construction algorithm for strings that we have introduced needs to be adjusted. After having discussed edit distance and suffix trees we describe how to use them to produce constituent hypotheses by comparing sentences. For the new suffix tree approach we find three variants with various generative capacity.

Both the edit distance alignment and the suffix tree alignment have been tested on three corpora: the ATIS, OVIS and WSJ corpus. For the last corpus, we have tested on section 23 and section 02-21+23. From the timing results in Chapter 5 we can conclude that suffix tree alignment allows systems such as ABL to learn on large corpora as well. The difference in theoretic complexity is confirmed by empirical results (see Section 5.6). For the ATIS, suffix tree alignment learning is a factor ~ 11 faster. On the larger OVIS, suffix tree alignment learning is a factor ~ 386 faster. However, it can be observed that for more complex corpora such as the WSJ, the recall of useful hypothesis constituents found seems to decrease with increasing corpus size: learning on section 23 results in 52.2 percent recall whereas learning on section 02-21+23 results in 37.3 percent recall.

6.2 Future work

Although the experimental results are very encouraging, there remain a few issues that deserve further research.

In the first place, the decrease in recall when learning on sections 02-21 (+23) of the WSJ corpus needs to be examined further. Despite the high number of hypotheses introduced, the recall is very low.

In the second place, it is interesting to test how the selection learning phase will perform with the hypotheses introduced by edit distance alignment learning compared with those introduced by suffix tree alignment learning.

In the third place, it would be interesting to see how the three suffix tree alignment methods would perform on corpora of left-branching languages such as Japanese. In the previous chapter, it was predicted that the method *suf*, which introduces only suffix hypotheses, would perform worse on aligning sentences in left-branching languages.

Furthermore, it is remarkable that the recall of edit distance alignment (default method) and suffix tree alignment (presuf 2 method) diverges considerably on learning over subsets of the OVIS corpus that increase in size. Apparently, there are syntactic constructs which are included in edit distance alignment and excluded in suffix tree alignment of vice versa. In either case, it seems that the generative capacity of both method is different and the number of constituents proposed show that the suffix tree method is in general more generative. In this line, it is interesting to consider the possibility of introducing parameters in the suffix tree alignment methods that can put restrictions on the expressiveness in order to pursue a situation in which a lowest number of hypotheses needs to be removed.

Finally, now we have seen the merits of the suffix tree data structure in finding regularities in corpora, it might be fruitful to consider the use of suffix trees in other grammatical inference systems.

Bibliography

- Adriaans, P. [1999]. Learning shallow context-free languages under simple distributions. Technical Report Tech. rep. ILLC Report PP-1999-13, Institute for Logic, Language and Computation, Amsterdam, the Netherlands.
- Adriaans, P., Trautwein, M. and Vervoort, M. [2000]. Towards high speed grammar induction on large text corpora. In Hlavac, V., Jeffery, K. G. and Wiedermann, J. (Eds.), *SOFSEM 2000: Theory and Practice of Informatics* (pp. 173–186). Springer-Verlag.
- Baker, J. K. [1979]. Trainable grammars for speech recognition. In Klatt, D. H. and Wolf, J. J. (Eds.), *Speech Communication Papers for the 97th Meeting of the Acoustic Society of America* (pp. 547–550). NY, USA: Acoust. Soc. Am.
- Boyer, R. S. and Moore, J. S. [1977]. A fast string search algorithm. *Communications of the Association for Computing Machinery*, 20(10), 762–772.
- Brill, E. and Marcus, M. [1992]. Automatically acquiring phrase structure using distributional analysis. In *Proceedings of the DARPA Workshop on Speech and Natural Language* (pp. 259–265). NY, USA: Harriman.
- Brown, R. and Hanlon, C. [1970]. Derivational complexity and order of acquisition in child speech. In J. R. Hayes (Ed.), *Cognition and the Development of Language* chapter 1, (pp. 11–54). New York, USA: John Wiley.
- Carroll, G. and Charniak, E. [1992]. Two Experiments on Learning Probabilistic dependency grammars from corpora. In Weir, C., Abney, S., Grishman, R. and Weischedel, R. (Eds.), *Working Notes of the Workshop Statistically-Based NLP Techniques* (pp. 1–13). American Association for Artificial Intelligence (AAAI).
- Charniak, E. [1993]. *Statistical Language Learning*, chapter Probabilistic Context-Free Grammars, (pp. 81–82). Cambridge, London, UK: MIT Press.
- Charniak, E. [1997]. Statistical parsing with a context-free grammar and word statistics. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence* (pp. 598–603). American Association for Artificial Intelligence (AAAI).

- Chomsky, N. [1957]. *Syntactic structures*. Janua linguarum. The Hague, the Netherlands: Mouton.
- Clark, A. S. [2001]. *Unsupervised Language Acquisition: Theory and Practice*. PhD thesis, University of Sussex.
- Collins, M. [1997]. Three generative, lexicalised models for statistical parsing. In *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics (ACL)* (pp. 16–23). Madrid, Spain: Association for Computational Linguistics (ACL).
- Finch, S., Chater, N. and Redington, M. [1995]. Acquiring syntactic information from distributional statistics. In J. P. Levy, D. Bairaktaris, J. A. Bullinaria and P. Cairns (Eds.), *Connectionist Models of Memory and Language* (pp. 229–242). London, UK: UCL Press.
- Gold, E. M. [1967]. Language identification in the limit. *Information and Control*, 10, 447–474.
- Grünwald, P. [1996]. Volume 1040 of *Lecture Notes in Artificial Intelligence*, chapter A Minimum Description Length Approach to Grammar Inference, (pp. 203–216). Berlin, Germany: Springer Verlag.
- Gusfield, D. [1997]. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge, UK: Cambridge University Press.
- Harris, Z. S. [1951]. *Methods in Structural Linguistics*. Chicago, USA: University of Chicago Press.
- Horning, J. J. [1969]. *A Study of Grammatical Inference*. PhD thesis, Stanford University.
- Huybregts, R. M. A. C. [1984]. The weak adequacy of context-free phrase structure grammar. In G. J. de Haan, M. Trommelen and W. Zonneveld (Eds.), *Van periferie naar kern* (pp. 81–99). Dordrecht, the Netherlands: Foris.
- Klein, D. and Manning, C. D. [2001]. Distributional phrase structure induction. In Daelemans, W. and Zajac, R. (Eds.), *Proceedings of the Fifth Workshop on Natural Language Learning (CoNLL 2001)* (pp. 113–120). Toulouse, France.
- Knuth, D. E., Morris, J. H. and Pratt, V. R. [1977]. Fast pattern matching in strings. *Society for Industrial and Applied Mathematics Journal on Computing*, 6, 323–350.
- Lamb, S. M. [1961]. On the mechanisation of syntactic analysis. In *1961 Conference on Machine Translation of Languages and Applied Language Analysis*, Volume 2 of *National Physical Laboratory Symposium No. 13* (pp. 674–685). London, UK: Her Majesty's Stationery Office.
- Lari, K. and Young, S. [1990]. The estimation of stochastic context-free grammars using the inside-outside algorithm. *Computer Speech and Language Processing*, 4(1), 35–56.

- Levenshtein, V. I. [1965]. Binary codes capable of correcting deletions, insertions and reversals. In *Doklady Akademii Nauk SRR*, Volume 163 (pp. 845–848). Original in Russian.
- Li, M. and Vitányi, P. M. B. [1991]. Learning simple concepts under simple distributions. *Society for Industrial and Applied Mathematics (SIAM) Journal on Computing*, 20(5), 911–935.
- de Marcken, C. G. [1996]. *Unsupervised Language Acquisition*. PhD thesis, Department of Electrical Engineering and Computer Science, MIT.
- Marcus, M. P., Santorini, B. and Marcinkiewicz, M. [1993]. Building a large annotated corpus of English: the Penn Treebank. *Computational linguistics*, 19, 313–330. Reprinted in Susan Armstrong, ed. 1994, *Using large corpora*, Cambridge, MA: MIT Press, 273–290.
- Masek, W. J. and Paterson, M. S. [1980]. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20, 18–31.
- McCreight, E. M. [1976]. A space-economical suffix tree construction algorithm. *Journal of the Association for Computing Machinery*, 23(2), 262–272.
- Mitchell, T. M. [1997]. *Machine learning* (pp. 2–3). New-York, USA: MIT Press, McGraw-Hill.
- Pereira, F. and Schabes, Y. [1992]. Inside-outside reestimation from partially bracketed corpora. In *Proceedings of the 30th Annual Meeting of the Association for Computational Linguistics (ACL)* (pp. 128–135). Delaware, USA: Association for Computational Linguistics (ACL).
- Shannon, C. E. and Weaver, W. [1949]. *The mathematical theory of communication*. Urbana, USA: University of Illinois Press.
- Shieber, S. M. [1985]. Evidence against the context-freeness of natural language. *Linguistics & Philosophy*, 8(3), 333–343.
- Sokolov, J. K. and Snow, C. E. [1994]. The changing role of negative evidence in theories of language development. In C. Gallaway and B. J. Richards (Eds.), *Input and Interaction in Language Acquisition* (pp. 38–55). New York, USA: Cambridge University Press.
- Solomonoff, R. [1964]. A formal theory of inductive inference. *Information and Control*, 7, 224–254.
- Stolcke, A. [1994]. *Bayesian Learning of Probabilistic Language Models*. PhD thesis, University of California, Berkeley, USA.
- Ukkonen, E. [1995]. On-line construction of suffix trees. *Algorithmica*, 14(3), 249–260.

- Valiant, L. G. [1984]. A theory of the learnable. *Communications of the Association for Computing Machinery*, 27(11), 1134–1142.
- Wagner, R. A. and Fischer, M. J. [1974]. The string-to-string correction problem. *Journal of the Association for Computing Machinery*, 21(1), 168–173.
- Weiner, P. [1973]. Linear pattern matching algorithms. In *Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory* (pp. 1–11). Los Alamitos, Calif., USA: IEEE Computer Society Press.
- Wolff, J. G. [1988]. Learning syntax and meanings through optimalization and distributional analysis. In Y. S. Levy and M. D. S. Braine (Eds.), *Categories and Processes in Language Acquisition* (pp. 85–98). Hillsdale NY, USA: Lawrence Erlbaum.
- van Zaanen, M. [2000]. ABL: Alignment-Based Learning. In *Proceedings of the 18th International Conference on Computational Linguistics (COLING)* (pp. 961–967). Saarbrücken, Germany: Association for Computational Linguistics (ACL).
- van Zaanen, M. [2002]. *Bootstrapping Structure into Language: Alignment-Based Learning*. PhD thesis, University of Leeds, Leeds, UK.
- van Zaanen, M. and Adriaans, P. [2001]. Alignment-Based Learning versus EMILE: A comparison. In *Proceedings of the Belgian-Dutch Conference on Artificial Intelligence (BNAIC); Amsterdam, the Netherlands* (pp. 315–322).