Two-level probabilistic grammars for natural language parsing

Infante Lopez, G.G.
Chapter 6
Splitting Training Material Optimally

6.1 Introduction

Our approach to parsing can be viewed as a simple CF parser with the special feature that our context free rules do not exist a priori. Instead, there is a device for generating them on demand. The device produces strings and these strings are used as CF bodies of rules. In the previous chapter, we used probabilistic automata for generating bodies of rules. These automata were not built manually, but we induced them from sample instances obtained from tree-banks. The general idea used for building the probabilistic automata model bodies of rules, consists of copying all bodies of rules inside the Penn Tree-bank (PTB) to a bodies of rules sample bag. This sample bag is treated as the sample set of a regular language and probabilistic automata are induced from it. Once the probabilistic automata have been built, they are used for defining a grammar that uses them for building rules on the fly.

The sample bag of rule bodies to be used as training material contains many different types of strings. For example, some strings may describe arguments of transitive verbs, while others may describe arguments of intransitive verbs. Chapter 5, together with previous work from the literature (Galen et al., 2004), suggests that models which are induced from the sample bag can be substantially improved as follows. First, split the training material into sets containing only homogeneous material; second, induce a model for each class, and third, combine the different models into one general model.

The assumption underlying this three step procedure is that the regular language we want to model is in fact the union of several languages. We split the material guided by the aim of keeping apart strings that belong to different languages. In this way, the models induced from each of the bags created after splitting the material are cleaner, because the algorithm that induces a regular language from one of the resulting training bags simply does not see any string belonging to other bags. By splitting the
data splitting we try to minimize the noise to which the learning algorithm is exposed.

To help shape our intuitions, let us consider two examples. First, suppose that the language we try to learn is the set \( \{a, aa, aaa, aaaa, aaaaa\} \). Suppose that we have a bag of instances of this language and that we want to infer the original language from it. The learning algorithm might consider the bag of samples as instances of the language \( a^* \), while if we split the sample bag into five different bags, each containing strings of the same length, it is more clear for the learning algorithm that each of the bags is produced by a language containing only one element.

Second, suppose that we want to model verb arguments. Simplifying, verb arguments can be thought of as the union of transitive verb arguments and intransitive verb arguments. Our working hypothesis is that a better model can be induced for verb arguments if we first split the training material into two different samples, one containing all the instances of transitive verbs and the other containing all the instances for intransitive verbs. Once the training material has been split, two models are induced, one modeling transitive verbs and the other modeling intransitive ones. The original material is divided into two bags to avoid the data instances from these two different phenomena from interfering with each other. Conceptually, if the data is not split, the algorithm for inducing a regular language for intransitive verbs sees the sample instances of transitive verbs as noise and vice-versa.

How do we split the training material? One possible way consists in defining a split by hand. Chapter 5 provides an example of this approach. There, the training material was split according to the head word of the dependency rules. As a consequence, two different automata for each part-of-speech (POS) were induced, one modeling right dependents and one modeling left dependents. In contrast, the approach we pursue in this chapter aims at finding an optimal splitting in an unsupervised manner. For this purpose we define a quality measure that quantifies the quality of a partition, and we search among a subset of all possible partitions for the one maximizing the proposed quality measure. Thus, one of our main challenges will be to find such a measure. Once the partition that optimizes our quality measure has been found, we use it for building as many automata as there are components in the partition. Finally, we use the induced automata for building PCW grammars, which we then use for parsing the PTB.

In this chapter we present a measure that quantifies the quality of a partition, we also show that the measure we found correlates with parsing performance. As a consequence, the procedure we use for splitting the material is a procedure that can be used for finding optimal grammars, optimal in the sense of parsing performance, without having to parse the PTB.

This chapter is organized as follows. Section 6.2 presents an overview of the chapter; Section 6.3 explains how to build grammars once the optimal partition has been
found; Section 6.4 explains how we search for the optimal partition, and Section 6.5 reports on the results on parsing the PTB. Section 6.6 discusses related work, and Section 6.7 states conclusions and describes future work.

6.2 Overview

We want to build grammars using training material that has been split into homogeneous classes of strings. Our main research goal is to understand how the elements in the training material interfere with each other, thus diminishing the quality of the resulting grammars. We also want to quantify the gain in terms of parsing performance that can be obtained by splitting the training material. Furthermore, we are interested in finding a quality measure for grammars that only takes the grammar’s structure into consideration and helps us to predict the grammar’s performance in parsing without actually parsing.

We proceed as follows. As in Chapter 5, we first transform the PTB into projective dependency structures following (Collins, 1996); see Section 2.1.1 for details. From the resulting tree-bank we delete all lexical information except POS tags. Every occurrence of a POS in a tree belonging to the tree-bank has associated to it two different, possibly empty, sequences of right and left dependents, respectively. We extract these sequences for all trees, producing two different bags containing right and left sequences of dependents, respectively.

We then proceed with a first splitting of the training material. For this purpose we use the POS tag of the head word as described in Chapter 5. This first splitting produces two different sample bags for each POS, one containing instances of left dependents, and the other containing instances of right dependents.

To keep our experiments focused we decided to split the training material of a single POS tag only: VB. VB is one of the POS with the highest value of perplexity (PP); experiments in Chapter 5 suggest that higher values of PP are due to the use of a single automaton for modeling different regular languages. Recall that, for instance, the values of PP drop considerably when the training material is split using the POS tag of the head word. Since the PP value associated to VB is one of the highest, VB seems to include words with substantially different behaviors, an intuition that is clearly confirmed by the literature (Levin, 1993; Merlo and Stevenson, 2001). We isolate the sentences containing the VB tag and see how dealing only with VB affects other tags.

The initial partition of the training set corresponding to the VB tag is split using syntactic information such as father tag, number of dependents, depth in the tree, etc. So, all instances in the training material that share the same feature are placed in the same bag. The initial partition aims at using external knowledge to split the material:
we try to characterize each of the resulting bags according to the output of the syntactic
information used for building the split.

Recall that each component in the partition is a set of strings. We use such sets
to build as many automata as there are components in the partition. For each of the
automata built, we compute its quality, and the quality of the partition is defined as
a combination of the qualities of those individual automata. Once the initial partition
has been defined, genetic algorithms are used for finding a merging of components
in the partition that optimizes the quality measure. Next, we use the optimal merging
found by the genetic algorithm for building a PCW-grammar (see Section 6.3). Finally,
we use the resulting grammar for parsing the PTB, and we report on the results in
Section 6.5.

6.3 Building Grammars

In order to build a grammar we need to complete five steps: (1) obtain the training
material from the PTB, (2) build an initial partition, (3) find an optimal partition con-
taining the initial partition, (4) induce an automaton for each component in the optimal
partition, and (5) put all automata together in a grammar. In this section we focus on
steps (1), (4), and (5), while Section 6.4 focusses on steps (2) and (3).

6.3.1 Extracting Training Material

We extracted the training and testing material from the PTB. As we did in Chapter 5, all
sentences containing CC tags are filtered out. We also eliminate all lexical information,
leaving POS tags only. Dependents are extracted from dependency trees. For each
dependency tree, we extract sample bags of right and left sequences of dependents. As
an example, the tree in Figure 6.1 is transformed into the dependency tree shown in
Figure 6.2. Its bags of left and right dependents are shown in Table 6.1.

From trees in sections 2–22 of the PTB we build two bags $T_L$ and $T_R$ containing left
and right dependents respectively. From trees in sections 0–1 we build two different
bags $Q_L$ and $Q_R$, also containing left and right dependents respectively. The bags $T_L$
and $T_R$ are used as training material for automata induction algorithms, while bags $Q_L$
and $Q_R$ are used for evaluating the resulting automata.

6.3.2 From Automata to Grammars

Let $T$ be a bag of training material extracted from the transformed tree-bank. Recall
from Section 2.2.2 that we use two different measures for evaluating the quality of
automata. Let $Q$ be a test bag extracted as $T$. We use perplexity (PP) and missed
6.3. Building Grammars

Figure 6.1: Tree extracted from the PTB, Section 02, file wsj_0297.mrg.

Figure 6.2: Dependency structure corresponding to the tree in Figure 6.1.
samples (MS) to evaluate the quality of a probabilistic automaton. A PP close to 1 indicates that the automaton is almost certain about the next step while reading the string. MS counts the number of strings in the test sample $Q$ that the automaton failed to accept.

Now, we describe how we build grammars once partitions over the bags of training material have been defined. Suppose that a partition $\Pi_{TVB} = \langle \pi_1, \ldots, \pi_n \rangle$ has been found over the training material $T_{VB}$. Suppose also that, for each component $\pi_i$ in the partition $\Pi_{TVB}$, two automata $A^L_{\pi_i}$ and $A^R_{\pi_i}$, modeling left and right dependents respectively, have been induced. Finally, suppose that there are two automata $A^L_w$ and $A^R_w$ for all POS $w$ in the PTB other than VB. Let $G^L_w$, $G^R_w$, $G^L_{\pi_i}$ and $G^R_{\pi_i}$ be their equivalent PCFGs obtained following (Abney et al., 1999). Let $S^L_w$, $S^R_w$, $S^L_{\pi_i}$ and $S^R_{\pi_i}$ be the start symbols of $G^L_w$, $G^R_w$, $G^L_{\pi_i}$ and $G^R_{\pi_i}$ respectively.

Our final grammar $G$ is defined as follows. Its start symbol is $S$, its set of pseudo-rules is defined as the union of

$$\{ W \xrightarrow{s} S^w w S^w, \quad S \xrightarrow{s} S^w w S^w : w \in \text{POS} \}$$

and

$$\{ VB^i \xrightarrow{s} S^i w BS^i, \quad S \xrightarrow{s} S^i w BS^i : \pi_i \in \Pi_{TVB} \},$$

and its set of meta-rules is the union of rules in $G^L_w$, $G^R_w$, $G^L_{\pi_i}$ and $G^R_{\pi_i}$ for all $w$ in POS and $\pi_i$ in $\Pi_{TVB}$. 

<table>
<thead>
<tr>
<th>Word Position</th>
<th>Word’s POS</th>
<th>Left</th>
<th>Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NN</td>
<td>NN</td>
<td>NN</td>
</tr>
<tr>
<td>1</td>
<td>MD</td>
<td>MD</td>
<td>MD VB DOTSYB</td>
</tr>
<tr>
<td>2</td>
<td>VB</td>
<td>VB</td>
<td>VB IN</td>
</tr>
<tr>
<td>3</td>
<td>IN</td>
<td>IN</td>
<td>IN NN</td>
</tr>
<tr>
<td>4</td>
<td>NN</td>
<td>NN</td>
<td>NN TO</td>
</tr>
<tr>
<td>5</td>
<td>TO</td>
<td>TO</td>
<td>TO VB</td>
</tr>
<tr>
<td>6</td>
<td>VB</td>
<td>VB</td>
<td>VB NN IN</td>
</tr>
<tr>
<td>7</td>
<td>DT</td>
<td>DT</td>
<td>DT</td>
</tr>
<tr>
<td>8</td>
<td>NN</td>
<td>NN</td>
<td>NN</td>
</tr>
<tr>
<td>9</td>
<td>IN</td>
<td>IN</td>
<td>IN NN</td>
</tr>
<tr>
<td>10</td>
<td>NN</td>
<td>NN</td>
<td>NN</td>
</tr>
<tr>
<td>11</td>
<td>NN</td>
<td>NN</td>
<td>NN</td>
</tr>
<tr>
<td>12</td>
<td>DOTSYB</td>
<td>DOTSYB</td>
<td>DOTSYB</td>
</tr>
</tbody>
</table>

Table 6.1: Bags of left and right dependents. Left dependents are to be read from right to left.
6.4 Splitting the Training Material

Let $T^R_{VB}$ and $T^L_{VB}$ be the training material corresponding to the right and left dependents of words tagged with VB. Let $Q^L_{VB}$ and $Q^R_{VB}$ be the left and right dependents from the tuning set whose head symbol is VB.

Let $\Pi = <\pi_1, \ldots, \pi_n>$ be a partition of the set $T = T^R_{VB} \cup T^L_{VB} \cup Q^L_{VB} \cup Q^R_{VB}$ (we denote the disjoint union of bags $X$ and $Y$ as $X \cup Y$). Since $\Pi$ is a partition of $T$, it induces a partition of each of the bags $T^R_{VB}$, $T^L_{VB}$, $Q^L_{VB}$ and $Q^R_{VB}$ when each of the sets is intersected with $\pi_i$. For example, the partition induced by $\Pi$ over $T_R$ is defined as $\Pi^R_{\pi_i} = (\pi_1 \cap T^R_{VB}, \ldots, \pi_n \cap T^R_{VB})$.

Once a partition of $T_R$ and $T_L$ is defined, constructing a grammar is straightforward. Now, we focus on how to construct partitions $\Pi$. Partitions $\Pi$ are defined in a twofold procedure. The first step defines an initial partition $\Pi = <\pi_1, \ldots, \pi_n>$ using syntactic features. Syntactic features help to group VB arguments according to the position in which they appear in the sentences extracted from the tree-bank. In the second step, a quality measure for partitions is defined and an optimization of the global quality of the partition is performed. The optimization phase searches for the optimal partition among all partitions containing the initial partition. Consequently, the initial partition determines the space search for the optimization phase in the second step.

6.4.1 Initial Partitions

In order to define initial partitions we use features. A feature is a function $f$ that takes two arguments; a dependency tree $t$ and a number $i$. The number $i$ is used as a reference to the $i$-th position in the sentence $x$ to which $f$ is applied. Since words in $x$ are in direct correspondence with the nodes of the tree $t$ yielding $x$, the index $i$ also corresponds to a node in the tree $t$. A feature returns any piece of information regarding the position of the index $i$ in the tree. Table 6.2 contains the features we use together with a brief description for each of them. For each feature $f$ in Table 6.2, the table's third column shows the result of applying $f$ to the tree in Figure 6.2 at position 2.

From a linguistic point of view, our features are used to characterize the dependents a verb might take. The underlying assumption is that features are capable of capturing the different behaviors that words tagged with VB might display. The idea is to group training instances according to their behavior. We divided the training material depending on the value a particular feature takes for a particular word in the particular tree where the word appears. We put all words' dependents tagged with VB with similar feature values into the same sample set. Consequently, the initial sample set is divided into smaller sample sets, each containing all dependents of words tagged VB.
Table 6.2: All features we use; they all take two arguments: a dependency tree $t$, and a node index $i$.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>WordStem</td>
<td>stem of the word at $i$</td>
<td>$\text{WordStem}(2) = \text{apply}$</td>
</tr>
<tr>
<td>gFather</td>
<td>the grand-father of $i$</td>
<td>$\text{gFather}(2) = \text{NN}$</td>
</tr>
<tr>
<td>Father</td>
<td>the father of $i$</td>
<td>$\text{Father}(2) = \text{IN}$</td>
</tr>
<tr>
<td>Depth</td>
<td>the depth of the tree below $i$</td>
<td>$\text{Depth}(2) = 1$</td>
</tr>
<tr>
<td>rSibling</td>
<td>first left sibling of $i$</td>
<td>$\text{rSibling}(2) = \text{NONE}$</td>
</tr>
<tr>
<td>FstLeftDep</td>
<td>the first left dependent of $i$</td>
<td>$\text{FstRightDep}(2) = \text{IN}$</td>
</tr>
<tr>
<td>NumLeftDep</td>
<td>the numbers of left dependents of $i$</td>
<td>$\text{NumRightDep}(2) = 1$</td>
</tr>
</tbody>
</table>

that share the same feature value. For example, suppose we use the feature $\text{father()}$ to partition the training material. All components in the partition share the same value of $\text{father}$ and there are as many components as there are possible outcomes for the feature $\text{father()}$. The underlying assumption becomes then, that all instances in a component are sampled from a regular language different from the regular language from which others components are sampled.

Formally, the initial partition is defined as follows. Let $T$ be the bag containing all training material; let $x$ be an element in $T$; let $t_x$ be the tree in the tree-bank from which $x$ was extracted. Let $i_x$ be the position in $t_x$ from which $x$ was extracted. Finally, let $f_1, \ldots, f_k$ be the sequence of features we want to use for defining an initial partition. The initial partition $\Pi = \langle \pi_1, \ldots, \pi_n \rangle$ is given as the equivalence classes defined by the following equivalence relation $R$:

$$ xRy \iff f_j(t_x, i_x) = f_j(t_y, i_y), j = 1, \ldots, k. $$

Once a feature has been defined, we have to assign new tags to all the material we used for building and testing the automata. For example, suppose that we use the $\text{father}$ feature to produce the initial splitting. In this case, the tree in Figure 6.2 is transformed into the tree in Figure 6.3. The training material related to the retagged VB tags is shown in Table 6.3.

![Figure 6.3: Dependency tree retagged according to the newly defined splitting.](image-url)
6.4. Splitting the Training Material

We use features to induce a partition of the training material and of the testing material. Since the testing material is much smaller than the training material this might yield empty components. Since the values for our quality measures obtained from empty components are meaningless, we merge those empty components with those where the resulting automaton has the lowest perplexity. The resulting partition has no empty component. Such a partition is the starting partition for the algorithm searching for the optimal merging. We present 8 different grammars built using different features, the features used are described in Table 6.4. This table also shows the number of components each feature produces together with the number of components in after having searched for the best partition.

<table>
<thead>
<tr>
<th>Grammar name</th>
<th># components in initial partition</th>
<th># components in optimal partition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>rSibling</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>NumRightDep</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Father</td>
<td>13</td>
<td>6</td>
</tr>
<tr>
<td>gFather</td>
<td>30</td>
<td>9</td>
</tr>
<tr>
<td>Depth</td>
<td>17</td>
<td>10</td>
</tr>
<tr>
<td>FstRightDep</td>
<td>27</td>
<td>17</td>
</tr>
<tr>
<td>WordStem</td>
<td>373</td>
<td>57</td>
</tr>
</tbody>
</table>

Table 6.4: Features used and the number of components in the partitions they induce.

6.4.2 Merging Partitions

In this section we discuss the algorithm that searches for partitions over the training material containing the initial partition. We say that a new partition $\Pi'$ contains a partition $\Pi$ if for any two elements $p$ and $q$ that belong to the same component $\pi_k$, there is a component $\pi'$ in $\Pi'$ such that both elements are again in $\pi'$. Our intention is to search among the partitions that contain the initial partition for an optimal one. In
order to decide which partition is the optimal one, we first need to define a measure for evaluating its quality.

Recall that a component in a partition is used to define an automaton. For each component in the partition, we can define a value of PP and MS; in what follows we use each of these individual values to define a quality measure for the whole partition.

For every candidate merging, and for computing the partition's quality, we have to assign new tags to all the material. We assign new tags according to the redefinition of the features function we used for building the initial partition. For example, suppose that the \texttt{father()} feature was used for building the initial partition, suppose also that a candidate merge states that the component where \texttt{father} is equal to \texttt{MD} should be merged to the component where \texttt{father} is equal to \texttt{TO}. All tags \texttt{VB} in the training material with fathers tagged \texttt{MD} and \texttt{TO} have to be retagged with the same tag. For example, the tree in Figure 6.2 becomes the tree in Figure 6.4, where tags \texttt{VB} are renamed as \texttt{VB-1}.

![Diagram](image)

Figure 6.4: Assigning new tags for computing the merging of \([\text{TO}]\) and \([\text{MD}]\).

Our measure has two main parts, each of which considers the automata related to the left and to the right side. In order to simplify the exposition, we describe in detail our measure for the component referring to the right side. The component referring to the left side is obtained by replacing \(R\) in the superscripts with \(L\).

Let \(\Pi = (\pi_1, \ldots, \pi_n)\) be a partition of the training material \(T\). Let \(A_i^R, i = 1, \ldots, k\) be the automata induced, as described in Section 6.3.2, using training sets \(\pi_i \cap T_R\), respectively. Let \(PP_i^R\) and \(MS_i^R\) be the values of PP and MS respectively for the automaton \(A_i^R\), computed using test sets \(\pi_i \cap Q_R\), for \(i = 1, \ldots, k\).

Our measure combines the values of \(PP_i\) and \(MS_i\) for all \(i\). That is, we combine all values of PP and MS to obtain a quality value of the whole partition. PP and MS values can not be summed up directly given that the importance of an automaton is proportional to the number of times it is used in parsing. The importance of PP and MS values should be proportional to the number of times the corresponding automaton is to be used in generating bodies of rules. We have estimates of such frequencies using the training material. For that purpose, let

\[
PP_i^R = \frac{|\pi_i \cap T_R|}{|T_R|}, i = 1, \ldots, n.
\]
One can view $p_i^R$ as the probability of using the automata $A_i^R$. We use these probabilities to measure the expected value for MS and PP as follows. Let $E[MS_{\Pi_i}^R]$ be the expected value of MS for a right automata defined as

$$E[MS_{\Pi_i}^R] = \sum_{i=1}^{n} p_i^R MS_i^R.$$  

Let $E[PP_{\Pi_i}^R]$ be the expected value of PP, defined as:

$$E[PP_{\Pi_i}^R] = \sum_{i=1}^{n} p_i^R PP_i^R.$$  

Let $E[MS_{\Pi_i}^L]$ and $E[PP_{\Pi_i}^L]$ be the corresponding expected values for the left sides. Note that the expected values depend on a particular partition, hence the subscript $\Pi$. We are now in a position to compare the quality of two partitions according to the values they assigned to $E[PP_{\Pi_i}^R]$, $E[MS_{\Pi_i}^R]$, $E[PP_{\Pi_i}^L]$ and $E[MS_{\Pi_i}^L]$. We say that partition $\Pi_1$ is better than partition $\Pi_2$ if all of the following holds:

$$E[PP_{\Pi_1}^R] < E[PP_{\Pi_2}^R], \quad \text{(6.1)}$$

$$E[MS_{\Pi_1}^R] < E[MS_{\Pi_2}^R], \quad \text{(6.2)}$$

$$E[PP_{\Pi_1}^L] < E[PP_{\Pi_2}^L], \quad \text{(6.3)}$$

$$E[MS_{\Pi_1}^L] < E[MS_{\Pi_2}^L]. \quad \text{(6.4)}$$

Ideally, we would like to find a quality function $q$ defined over the class of possible partitions such that $q(\Pi_1) < q(\Pi_2)$ if and only if Equations 6.1 through 6.4 are satisfied. If such a function exists, we can use many optimization methods for finding the partition for which $q$ is minimal. But, it is easy to see that such a function does not exist. In what follows, we show that even a function $q$ satisfying

$$q(\Pi_1) < q(\Pi_2) \iff (E[PP_{\Pi_1}^R] < E[PP_{\Pi_2}^R]) \land (E[MS_{\Pi_1}^R] < E[MS_{\Pi_2}^R]) \quad \text{(6.5)}$$

does not exist. Suppose that such a function does exist; suppose that the partitions $\Pi_1$ and $\Pi_2$ are two possible partitions with values $E[PP_{\Pi_1}]$, $E[MS_{\Pi_1}]$, $E[PP_{\Pi_2}]$, $E[MS_{\Pi_2}]$ for PP and MS. In order to compare the pair ($E[PP_{\Pi_1}]$, $E[MS_{\Pi_1}]$) with the pair ($E[PP_{\Pi_2}]$, $E[MS_{\Pi_2}]$) we can plot each pair as a vector, as shown in Figure 6.5. Since $q$ is defined for all partitions, it takes values $q_1 = q(\Pi_1)$ and $q_2 = q(\Pi_1)$. Both $q_1$ and $q_2$ are real numbers, so $q_1 \leq q_2$ or $q_2 \leq q_1$, both possibilities imply that $q(\Pi_1) \leq q(\Pi_2)$ or $q(\Pi_2) \leq q(\Pi_1)$, which contradicts Equation 6.5 if $q_1 \neq q_2$. The constraints imposed by Equation 6.5 are impossible to fulfill because they required function $q$ to map a partial order, defined over pairs of reals in the right-hand side to a total order defined over reals in the left-hand side. We can not apply a function minimization
algorithm to minimize both $E[PP]$ and $E[MS]$ at the same time because they can not be combined in one function that minimizes both whenever the one function is minimized. Still, we want to minimize both values at the same time. We propose to circumvent this problem by fixing a reference point and biasing a function optimization algorithm to improve over the reference point. We try to optimize a given starting configuration of $E[PP_\Pi]$, $E[MS_\Pi]$, $E[PP_{\Pi_0}]$ and $E[MS_{\Pi_0}]$. We explain our algorithm for $E[PP_{\Pi_0}]$ and $E[MS_{\Pi_0}]$, extending it to the four components is straightforward. Suppose that our given reference point $\Pi_0$ is the one described by Figure 6.6. All vectors that satisfy Equation 6.5 are vectors inside the shaded area. In order to improve over the values of PP and MS defined by vector $\Pi_0$ we have to search for those vectors that lay in the shaded area. Observe that every vector in the shaded area codifies the

![Figure 6.6: Values of $E[PP]$ and $E[MS]$ for two different partitions.](image)

quality of a partition. Since not all the vectors in the shaded area define the same quality, we need to define a measure to pick the optimal one from the shaded area. We use the norm of the vector as a quality measure because it tries to minimize all the
components at the same time. Formally, the function \( q_{\Pi_0} \) that we minimize is defined as follows:

\[
q_{\Pi_0}(\Pi) = \begin{cases} 
\|X\| + C & \text{if } E[PP_{\Pi_1}] > E[PP_{\Pi_0}], \\
\|X\| + C & \text{if } E[MS_{\Pi_1}] > E[MS_{\Pi_0}], \\
\|X\| + C & \text{if } E[PP_{\Pi_1}] > E[PP_{\Pi_0}], \\
\|X\| & \text{otherwise},
\end{cases}
\]

where \( X = (E[PP_{\Pi_1}], E[MS_{\Pi_1}], E[PP_{\Pi_1}], E[MS_{\Pi_1}]) \), \( C \) is a constant number and \( \|(x_1, x_2, \ldots, x_n)\| = \sqrt{x_1^2 + x_2^2 + \ldots + x_n^2} \). We use the constant \( C \) to penalize the vectors outside the shaded area of Figure 6.6. We drop the reference partition subscript from \( q \) whenever the reference vector is clear from the context.

The measure defined this way, is a measure that depends on a starting configuration. In Figure 6.7 we show an example in which \( q_{\Pi_0}(\Pi_2) < q_{\Pi_0}(\Pi_1) \), while \( \Pi_1 \) and \( \Pi_2 \) are incomparable using measures \( q_{\Pi_1} \) and \( q_{\Pi_2} \). It is also interesting to note that whenever partitions \( \Pi_0 \), \( \Pi_1 \) and \( \Pi_2 \) are such that \( q_{\Pi_0}(\Pi_1) < q_{\Pi_0}(\Pi_0) \) and \( q_{\Pi_1}(\Pi_2) < q_{\Pi_1}(\Pi_1) \) then \( q_{\Pi_0}(\Pi_2) < q_{\Pi_0}(\Pi_1) \).

In our experiments we use the trivial partition, i.e., the partition containing one and only one component containing all the training material, as reference point. Note that the reference point coincides with the partition we use in Chapter 5, consequently the results presented in this chapter are comparable to the experiments we performed in that chapter.

![Figure 6.7: Two incomparable solutions.](image)

We now apply this optimization technique to our specific problem, i.e., to find a merging of components that minimizes values of \( E[MS] \) and \( E[PP] \). We apply the procedure to different initial partitions: Table 6.5 shows the values of \( E[PP], E[MS], E[PS], E[MS], \) and \( q \) for all the grammars we build. Since all the experiments we carried out share the same reference partition, their values of \( q \) are comparable.
Chapter 6. Splitting Training Material Optimally

<table>
<thead>
<tr>
<th>grammar</th>
<th>left</th>
<th>right</th>
<th>q</th>
<th>alpha</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>1.189</td>
<td>0</td>
<td>9.633</td>
<td>6</td>
</tr>
<tr>
<td>rSibling</td>
<td>1.189</td>
<td>0</td>
<td>9.633</td>
<td>6</td>
</tr>
<tr>
<td>NumRightDep</td>
<td>1.189</td>
<td>0</td>
<td>9.633</td>
<td>6</td>
</tr>
<tr>
<td>Father</td>
<td>1.188</td>
<td>0</td>
<td>9.743</td>
<td>5</td>
</tr>
<tr>
<td>gFather</td>
<td>1.189</td>
<td>0</td>
<td>9.652</td>
<td>6</td>
</tr>
<tr>
<td>Depth</td>
<td>1.180</td>
<td>1</td>
<td>9.783</td>
<td>5</td>
</tr>
<tr>
<td>FstRightDep</td>
<td>1.188</td>
<td>0</td>
<td>2.950</td>
<td>1.431</td>
</tr>
<tr>
<td>WordStem</td>
<td>1.195</td>
<td>0</td>
<td>9.647</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 6.5: Results on $q$ for all the grammars we built.

From Table 6.5 we can see that the rSibling and NumRightDep do not suggest any partition that outperforms the value of $q$ of the baseline; for this particular case genetic algorithms exhaustively searched the whole space of possibilities. This was possible because the space itself is not very big. The number of components in Table 6.4 gives a hint about the size of the space of possible merges.

To understand the meaning of the measure $q$ in the context of two-level parsing, it is important to recall from Chapter 5 the meaning of PP and MS in this context. Recall that we are building PCW-grammars, and parsing with such grammars can be viewed as a two-phase procedure. The first phase consists in creating the rules that will be used in the second phase. The second phase consists in using the rules created in the first phase as a PCFG and in parsing the sentence using a PCF parser. Since automata are used to build rules, the values of PP and MS quantify the quality of the set of rules built for the second phase: MS gives us a measure of the number of rule bodies that should be created but that will not be created, and, hence, it gives us an indicator of the number of "correct" trees that will not be produced. PP tells us how uncertain the first phase is about producing rules. Now, $q$ tries to minimize these two aspects: a partition that outperforms the baseline means that the automata we induced missed, on average, a smaller number of bodies of rules and that the bodies of rules that are created, on average, are created with lower perplexity.

Searching for the Optimal Partition

Let $\Pi$ be an initial partition of $T$ built as described in Section 6.4.1. Let $\Pi_0$ be the reference partition, i.e., the partition containing one component in which the whole of the training material is found. For each of the initial partitions we defined in Section 6.4.1, we search for the merging that optimizes the quality function $q$.

Formally, the search space is defined as the set of possible partitions containing the
6.4. Splitting the Training Material

initial one. Let II and II' be two partitions over $T$, and let $a$ and $b$ be two elements in $T$. Recall that a partition II contains another partition II' if all components in II result from merging components in II'. Consequently, a partition containing II' can be easily generated by merging some of its components.

In order to search for the partition giving the minimum value of $q$ we use Genetic Algorithms (GAs). We use GAs because our problem can naturally be re-phrased as a GA optimization problem.

In GAs, a population of individuals competes for survival. Each individual is designated by a bag of genes that define its behavior. Individuals that perform better (as defined by the fitness function) have a higher chance of mating with other individuals. A GA implementation runs for a discrete number of steps, called generations. What happens during each generation can vary greatly depending on the strategy being used. Typically, a variation of the following happens at each generation:

1. **Selection.** The performance of all the individuals is evaluated based on the fitness function, and each is given a specific fitness value. The higher the value, the bigger the chance of an individual passing its genes on to future generations through mating (crossover).

2. **Crossover.** Selected individuals are randomly paired up for crossover (also known as sexual reproduction). This is further controlled by the crossover rate specified and may result in a new offspring individual that contains genes from each of its parents. New individuals are injected into the current population.

3. **Mutation.** Each individual is given the chance to mutate based on the mutation probability specified. Each gene of an individual is looked at separately to decide whether it will be mutated or not. Mutation is decided based upon the mutation rate (or probability). If a mutation is to happen, the value of the gene is switched to some other possible value.

In order to use GAs for our purposes we have to provide the following:

1. **A definition of individuals:** We design our individuals to codify two things. First, a value of alpha to be used for building the automata and second a partition of the training material. Alpha is simply codified as the first gene in the vector; the partition containing the initial partition is codified as follows. Note that in order to describe a partition for the training material it is enough to describe a way to merge components in the initial partition. Individuals in the population specify a way to merge components belonging to the initial partition into new components. A number $k$ in the $i$-th position in the vector $V$ indicates that component $i$ in the original partition should be added to the new component.
Formally, let $V = \langle a_1, \ldots, a_n \rangle$ be a vector with $i \leq a_i \leq n$. The vector $V$ defines the partition $\Pi' = \langle \pi'_1, \ldots, \pi'_m \rangle$ such that $\pi'_i = \bigcup \{ \pi_k : V[k] = i \}$.

Intuitively, the number of components in the resulting partition is equal to the number of different values stored in vector $V$. E.g., if all entries in $V$ are the same, the new partition defined by $V$ contains only one component.

2. **A fitness function defined on individuals**: The fitness function for an individual is defined as the quality measure $q$ we defined in Section 6.4.2.

3. **A strategy for evolution**: The strategy we follow is defined as follows. We apply two different operations to genes, namely crossover and mutation. We decide which operation to apply by flipping a biased coin. Crossover gets 0.95 probability of being applied while mutation gets 0.05. Once the operation is chosen, genes to which the operations apply are selected from the population. We select individuals using the *roulette wheel strategy* (Gen and Cheng, 1997), in which the probability for an individual to be selected is proportional to its fitness score. Crossover is implemented as follows, two points are selected along the chromosomes of both parents. The chromosomes are then cut at those points, and the middle parts are swapped, creating two child chromosomes. If mating occurs, two new genes are added to the population. If no mating occurs, no new gene is added to the population. Mutation is implemented as follows. Each gene of an individual is looked at separately to decide whether it will be mutated or not. Mutation is decided based upon the mutation rate (or probability). If a mutation is to happen, then the value of the gene is switched to some other possible value. For further details on the implementation of GAs we used see (Qumsieh, 2003)

Finally, the population of each of our generation consists of 50 individuals; we let the population evolve for 100 generations. We decided to use 100 generations because the computation of the quality of partition $q$ is time consuming and, moreover the quality measure and the number of partition are stable around generation 65 as pictured in Figure 6.8.

### 6.5 Parsing the Penn Treebank

Finally, we report on the accuracy in parsing. As in Chapter 5 we use two measures, %Words and %Pos. The former computes the fraction of the words that have been attached to their correct father, the latter computes the fraction of words that were attached to the correct word-class. As explained in Chapter 5 the two measures try to capture the performance of the PCW-parser in the two phases procedure described in
6.5. Parsing the Penn Treebank

Evolution of the fitness function

(a)

Evolution of the number of partitions

(b)

Figure 6.8: (a) The value of the quality measure $q$ across generations for 7 different grammars. (b) The number of components across generations. In both plots, plotted values at population $i$ correspond to the value of $q$ and the number of components of the individual with the highest $q$ among all individuals in $i$.

Chapter 5: (%POS) tries to capture the performance in the first phase, and (%Words) in the second phase.

The measures reported in Tables 6.6 and 6.7 are the mean values of (%POS) and (%Words) computed over all sentences in section 23 having length at most 20. We parsed only those sentences because our baseline, coming from Chapter 5, was computed on these sentences. The grammar names in Tables 6.6 and 6.7 refer to the features used for splitting the training material and building the grammars; see Table 6.2 for an explanation of each feature.

Since in our approach we split only the training material for VB we start by commenting the scores on parsing sentences containing the VB tag. Table 6.6 columns 4 and 5 shows the results. We have identified two sets of grammars. For those in the up-

<table>
<thead>
<tr>
<th>Grammar</th>
<th>$q$</th>
<th># components</th>
<th>with VB tag</th>
<th>without VB tag</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>%Words</td>
<td>%POS</td>
</tr>
<tr>
<td>Baseline</td>
<td>10.911</td>
<td>1</td>
<td>0.8484</td>
<td>0.8738</td>
</tr>
<tr>
<td>rSibling</td>
<td>10.911</td>
<td>1</td>
<td>0.8484</td>
<td>0.8738</td>
</tr>
<tr>
<td>NumRightDep</td>
<td>10.911</td>
<td>1</td>
<td>0.8484</td>
<td>0.8738</td>
</tr>
<tr>
<td>Father</td>
<td>9.432</td>
<td>6</td>
<td>0.8601</td>
<td>0.8846</td>
</tr>
<tr>
<td>gFather</td>
<td>9.413</td>
<td>9</td>
<td>0.8502</td>
<td>0.8758</td>
</tr>
<tr>
<td>Depth</td>
<td>9.065</td>
<td>10</td>
<td>0.8608</td>
<td>0.8855</td>
</tr>
<tr>
<td>FstRightDep</td>
<td>3.337</td>
<td>17</td>
<td>0.8623</td>
<td>0.8876</td>
</tr>
<tr>
<td>WordStem</td>
<td>4.705</td>
<td>57</td>
<td>0.8498</td>
<td>0.8758</td>
</tr>
</tbody>
</table>

Table 6.6: Results on parsing sentences containing and not containing the VB tag.
per part of the table there is a correlation between the quality measure and the parsing score given that for lower values of \( q \) we obtain greater values of parsing scores. This does not seem to be the case for the grammar in the lower part of the table. For the latter, even though the quality measure \( q \) is quite a bit smaller than the baseline (-4.705 vs. 10.911), its parsing score is only marginally different from the baseline score. As \( q \) gives us the expected values of PP and MS, intuitively this states the grammar in the lower part should have less ambiguity when choosing dependents of words tagged VB, but it produces virtually identical parsing scores even though its \( q \) values are lower than for nearly all grammars in the upper part. We think that is due to the number of components in the partitions. Note that the grammars in the upper part are induced from partitions containing less than 20 components, while the grammar in the lower part is induced from a partition containing no less than 57 components. As a consequence of having a large number of components each component contains a small amount of training and testing material and the values of PP and MS become unreliable. A better version of \( q \) should take into consideration the number of components in each partition and it should punish those components containing a small number of instances. Obviously, another way to overcome this problem is to use more training and testing material. Alternatively, a possible solution is to adopt a similar approach to the use in many clustering techniques where the number of components is required to be fixed and to search for the partition that optimizes \( q \) among all partitions having a fixed number of components.

With the experiments we carried out, it is possible to draw some conclusions for the grammars in the upper part of Table 6.6. Note that for those grammars, the ranking correlates (more correctly: inversely correlates) with the parsing score. This suggests that \( q \) is an indicator of the parsing score and that \( q \) can be used to quantify the quality of grammars without having to parse the whole gold standard. For the grammars in the upper part of Table 6.6, columns 4 and 5, we computed Pearson's product-moment correlation (NIST, 2004; Wright, 1997). We computed the correlation between \( q \) and %Words and between \( q \) and %POS for the case of sentences containing the VB tag. Pearson's correlation coefficient is usually signified by \( \rho \), and can take on the values from -1.0 to 1.0. Here, -1.0 is a perfect negative (inverse) correlation, 0.0 is no correlation, and 1.0 is a perfect positive correlation. The statistical significance of \( \rho \) is tested using a t-test. The t-test returns a p-value, where a low p-value (less than 0.05 for example) means that there is a statistically significant relationship between the two variables.

Now, Pearson's product-moment correlation test shows a correlation value of \( \rho = -0.821, p = 0.04484 \) and \( \rho = -0.835, p = 0.03832 \), for \( q \) vs %Words and \( q \) vs %POS, respectively. The correlation values suggest that \( q \) is a measure that only takes into consideration the way a grammar was built in order to predict its parsing performance.
Note that the values of $p$ are small, they are below 0.05 which is usually the weakest evidence that is normally accepted in experimental sciences. However, the correlation was computed only on a few sample points; in order to get more reliable values of correlation it is necessary to use bigger collections as training material and to define and compute $q$ for a larger number of grammars. Nevertheless, the correlation values found suggest that the observed differences are significant and that they are not the product of a random improvement.

Since all the training material is retagged according to the components induced for VB, automata induced for POS other than VB might alter their quality. In order to get a quantitative picture of the impact of the splitting in POS other than VB we separately parse sentences that do not contain the VB. Columns 6 and 7 in Table 6.6 show the results. Parsing scores are close to the baseline, whenever the sentences do not contain the VB tag. Phrased more positively, while optimizing for sentences containing words tagged VB, parser performance on sentences not containing words tagged VB did not decrease. Indeed, the Pearson's product-moment correlation tests for these columns show a correlation value of $\rho = -0.2860$, $p = 0.5816$ and $\rho = -0.5369$, $p = 0.271$, for $q$ vs %Words and $q$ vs %POS, respectively. These correlation values suggest that $q$ does not (inversely) correlate with parsing performance for sentences not containing the VB tag.

We can speculate that splitting material for one particular POS tag does not hurt the parser performance on other POS tags. This suggests that we could proceed by splitting different POS separately and then combine them in one grammar. The parsing performance for the final grammar should gain from all the gains in performance for each of the non-trivial splitting.

Finally, for the sake of completeness Table 6.7 presents the parsing scores for all the sentences in the test set. Observe that, indeed, the scores over all sentences do improve, even if we only optimized for a single POS.

<table>
<thead>
<tr>
<th>Grammar</th>
<th>$q$</th>
<th>#components</th>
<th>%Words</th>
<th>%POS</th>
<th>%POS - %Words</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>10.91</td>
<td>1</td>
<td>0.8491</td>
<td>0.8787</td>
<td>0.0296</td>
</tr>
<tr>
<td>rSibling</td>
<td>10.91</td>
<td>1</td>
<td>0.8491</td>
<td>0.8787</td>
<td>0.0296</td>
</tr>
<tr>
<td>NumRightDep</td>
<td>10.91</td>
<td>1</td>
<td>0.8491</td>
<td>0.8787</td>
<td>0.0296</td>
</tr>
<tr>
<td>Father</td>
<td>9.43</td>
<td>6</td>
<td>0.8525</td>
<td>0.8826</td>
<td>0.0301</td>
</tr>
<tr>
<td>gFather</td>
<td>9.41</td>
<td>9</td>
<td>0.8484</td>
<td>0.8787</td>
<td>0.0303</td>
</tr>
<tr>
<td>Depth</td>
<td>9.06</td>
<td>10</td>
<td>0.8545</td>
<td>0.8836</td>
<td>0.0291</td>
</tr>
<tr>
<td>FstRightDep</td>
<td>3.33</td>
<td>17</td>
<td>0.8550</td>
<td>0.8846</td>
<td>0.0296</td>
</tr>
<tr>
<td>WordStem</td>
<td>4.70</td>
<td>57</td>
<td>0.8494</td>
<td>0.8801</td>
<td>0.0307</td>
</tr>
</tbody>
</table>

Table 6.7: Results on parsing the PTB (all sentences).
6.6 Related Work

There are several perspectives from which we can analyze the approach and the experiments in this chapter. A first analysis sees our procedure to find optimal grammars as a way to induce preterminal symbols from the PTB. They are optimized for parsing and they provide information about the behavior of words tagged with VB. These preterminal symbols define a classification of verbs based on their syntactic behavior. There is a large collection of work on classification of verbs; most of them try to induce a classification of verbs using syntactic features, in some cases the resulting classification is evaluated (Merlo and Stevenson, 2001; Stevenson and Merlo, 2000), while in others the resulting classification (Decadt and Daelemans, 2004) is compared to the hand-crafted classification made by Levin (1993). We ran some experiments on trying to classify verbs according to the components they belong to. We checked the match between our classification and Levin’s manual classification of verbs. Unfortunately, we did not see any clear match between the two. We also explored manually the classification induced by our procedure, but we could not detect any linguistic explanation of the classification. We think that in order to get a linguistically meaningful classification, more training material and material tagged with other verb tags should be used.

A second analysis sees our procedure as a method for finding labels for estimating better probabilities. We can replace words by more general categories, like POS tags, in order to induce better parameters. The clusters we found can be viewed as new labels because these labels group words having comparable syntactic behavior. To use our labels in order to obtain better probabilities we need to retag not only the training material but also the gold standard. When the gold standard is retagged new tags codify some structural information. The whole approach is a simplified version of supertagging (Joshi and Srinivas, 1994; Srinivas, 1997) for PCW-grammars.

A third analysis considers the procedure as a way to induce sub-categorization frames (Manning, 1993; Carroll and Fang, 2004) for words tagged with VB. Our sub-categorization frames have the peculiarity that there is an infinite number of them given that each string accepted by our automata is a possible sub-categorization frame. Our induced sub-categorization frames are used for improving the parsing performance and are induced specially for this purpose. Only recently (Carroll and Fang, 2004; Yoshinaga, 2004; Hara et al., 2002) some work appeared where the induced sub-categorization frames are used for improving the parsing task.

Outside the context of parsing, the methodology we presented in this chapter can also be used for inducing regular languages. The idea of using clustering before inducing automata is not new. Dupont and Chase (1998) clustered symbols using standard clustering techniques (Brown et al., 1992; Ney and Kneser, 1993) before inferring the
automata. The main difference between our approach and theirs is that our algorithm presupposes that the target language is the union of different languages and the method tries to automatically detect the different components. Our algorithm also tries to detect the number of components automatically, while in (Dupont and Chase, 1998) the number of components is a parameter of the algorithm. From a more technical point of view, and still within the setting of inducing probabilistic regular languages, the procedure of, first, splitting the training material and, second, inducing as many regular languages as there are components, is a technique that guides the merging of states in the MDI algorithm and that disallows some of the possible merging. Recall from Section 2.2.2 that the MDI algorithm builds an automata, first, by building an initial automaton and, second, by merging some of the states in the initial automaton. When the material is split, not one but many initial automata are built. For this case, the MDI algorithm searches for candidate merges within each of the initial automaton. As a consequence, some candidate merges that were possible when inducing one automaton are not available any more in the case of many automata. Clearly, there are two questions that the splitting approach has to address, the first one is how to recombine the different automata in one single regular language, and the second, what criteria should we follow for splitting the training material? For our particular case, we use PCW-grammars for recombining the automata and syntactic information for splitting the training material.

From the point of view of optimization, we present a solution for optimizing two functions at the same time. The problem of optimizing more than one function at the same time is known as multiobjective optimization (Coello Coello, 1999). Briefly, multiobjective optimization techniques try to optimize a combination of many functions, called objectives, by finding a trade-off between the objectives. Under the multiobjective optimization perspective, it is possible to optimize the combination of objectives by optimizing one of them while others are not optimized.

### 6.7 Conclusions and Future Work

We presented an approach that aims at finding an optimal splitting of the training material, which in turn, is meant for improving parsing performance. For this purpose we defined a quality measure that quantifies the quality of partitions. Using this measure, we search among a subset of all possible partitions for the one maximizing the proposed quality measure. Our measure combines a quality measure defined for each component in a partition. To measure each component’s quality, we compute an automaton for each of component and we computed the automaton’s MS and PP. The measure we presented combines values of PP and MS for all resulting automata, one
Chapter 6. Splitting Training Material Optimally

per component, and it uses the resulting components to build grammars that are subsequently used for parsing the PTB.

For our particular case, it is not clear how to combine the functions \( E[PP] \) and \( E[MS] \) in such a way that the optimization of the combined values produces better parsing scores. What we know, is that if we optimize the four values \( (E[PP] \) and \( E[MS] \) for the left and right side) at the same time, we gain in parsing performance. While searching for the definition of the optimal function \( q \), we noticed that there might be a measure that uses only a subset of these four values. It seems that the four functions are not fully independent but the underlying relation remains an open problem.

We have shown that the quality measure we defined can be used for comparing parsing scores of two grammars whenever the grammars are built from partitions having a similar number of components. It would be interesting to define a measure that correlates with parsing performance independent of the number of components in the partition. The natural next step is, then, to define a measure that takes into account the number of components and the number of elements in each component. It is also important to investigate the impact a bigger corpus has in the measure we defined.

In this chapter we used PCW-grammars as the backbone for our experiments. They provide us with the appropriate level of abstraction for carrying out the experiments, and an easy way to combine all automata we induced for the different components into one single grammar. In contrast to the grammars in Chapter 5, the grammars we built in this chapter are not bilexical grammars. But, since the parser we implemented (see Appendix A for details) is a parser for PCW-grammars, it can handle both types of grammars. Grammars in Chapter 5 and the grammars in this chapter have in common that they search for unlabeled links.

Finally, this chapter changes the way the parsing task is usually addressed. Parsing is usually treated more as a modeling task than as an optimization task. A modeling task is a task where a model is designed and its parameters estimated from training material. Once these parameters are estimated, the model is tested on the parsing task and its results reported. In contrast, an optimization task is a task where a model is designed and its parameters are optimized according to the performance of the model and parameters in the final task. The difficulty of treating parsing as an optimization task resides in the time it takes to test a set of candidate values for the parameters. Since our measure \( q \) is a good indicator of the parsing performance we can treat parsing as an optimization task without having to parse. As a consequence, the procedure we defined is a procedure for building optimal grammars.