Experiments in Parsers Combination

Mémoire de DEA

Cécile Grivaz
Thesis Supervisor: Paola Merlo

June 12, 2007

Département de Linguistique
Université de Genève

Contents

List of Figures

Acknowledgements

I would like to express my gratitude to my supervisor, Paola Merlo. I'm deeply grateful for her constant guidance and her availability, for all the times she read and reread this thesis, and for being patient and exacting.

I would like to thank Gabriele Musillo for his much needed scientific support and great conversation.

I am also grateful to my friends Elisa Laurenti, Stéphane Magnenat, Emmanuel Eckard and Cyrille Dunant for scientific and moral support and for reading and correcting this thesis.

This thesis was partly funded by the Hasler Foundation.

Chapter 1

Introduction and Motivations

1.1 Introduction

Natural language processing is a field of computer science and linguistics that studies the automatic annotation, generation and understanding of written or spoken corpora in a human language such as English. Some of the important fields of natural language processing are information extraction which consists in finding specific information in natural language texts, information retrieval: finding among a large ensemble of documents the ones that correspond to a user request, and automatic translation. There are some common components among nearly every natural language processing application. One of them is the *parser*. A parser is able to find automatically the inner syntactic structure of a sentence. For example, consider the sentence:

1. Ada had tea with Charles

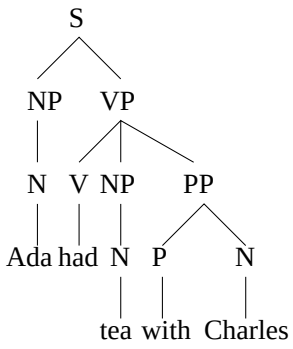


Figure 1.1: A syntactic tree. Note that the leaves represent the words of the sentence, and that the unary nodes over these leaves represent the part-of-speech tags of those words.

This sentence can have the underlying tree structure showed in figure 1.1. The leaves represent the words of the sentence. The lowest nodes over the leaves are always unary. The label of these pre-terminal nodes is the *part-of-speech tag* of the word that they dominate. We can classify each word of a natural language into a category. This category is the part-of-speech tag of the word.

Possible syntactic structures are described by a *grammar*. A grammar consists of a set of rewriting rules. The type of grammars considered in this thesis, consists of rules that allow us to build a certain type of structure: trees. A rule can be of the following form:

$$\mathrm{PP} \longrightarrow \mathrm{P\ N}$$

In sentence 1, this rule describes the subtree associated with the constituent *with Charles*. The left handside (*PP*: preposition phrase) represents the parent node, whereas the rights handside represents the children nodes (*P*: preposition and *N*: noun). Figure 1.2 shows the corresponding tree.

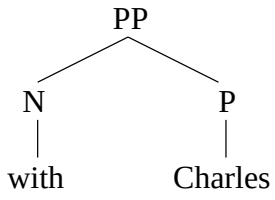


Figure 1.2: A constituent spanning the preposition phrase *with Charles*.

Parsing a *natural* language, such as English, is a difficult task. Specifically, it is much more difficult than parsing an *artificial* language, such as `Perl`. Parsers can also be useful for artificial languages. For example, a `Perl` compiler needs a parser. But the whole structure of an artificial language is well known, its vocabulary is small, and, most of all, it allows no *ambiguity*: a given sentence can be parsed with only one possible tree. Natural languages, however, are not controlled; their grammar can be very complex and ambiguous. Consider, for example, the following famous sentence:

2. Time flies like an arrow.

This sentence is highly ambiguous. Figure 1.3 shows some possible parse trees for this sentence. The first tree represents the usual interpretation of the sentence: a metaphor in which time is portrayed as a flying arrow. The interpretation associated with the second tree could be rephrased as: *time flies* (a fiction kind of insect) appreciate an arrow. The third tree represents

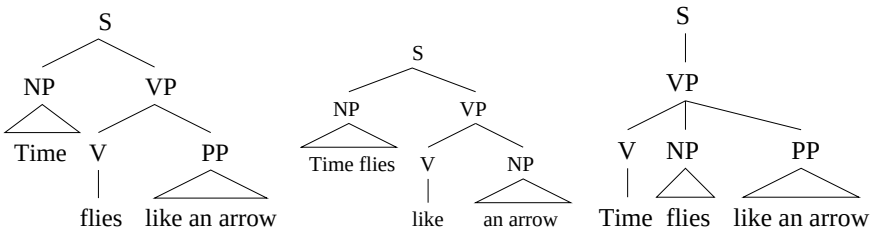


Figure 1.3: Possible parse trees for an ambiguous sentence. Note that each tree represents a different interpretation of the sentence.

an imperative sentence: time insects like you would time an arrow. The first interpretation would be semantically preferred, but the sentence is ambiguous at the syntactic level. The problem of ambiguity arises very often in natural languages, and makes the parsing task more difficult.

A syntactic analysis is the base for any higher level process, such as semantic or pragmatic analysis. Conversely semantic analysis can help in disambiguating possible syntactic trees, but a previous basic level of parsing is necessary for the semantic analysis. Consider for example the task of *semantic role labelling*, which is a basic semantic analysis. Surdeanu and Turmo (2005) achieved promising results on this task, when using a full rather than partial syntactic analysis.

Many *machine translation* systems use a parser. Liu and his colleagues (2006), for example, proposed a Chinese-English translation system based on *tree to string* alignment. In this model, a sentence from the source language is parsed, before being transformed into a string from the target language. Their model improved translation quality significantly, compared with a system based on phrases.

Speech to text also can use syntactic analysis to remove ambiguity. Speech to text applications often recognise several possible sequences of words in the input signal. They must then compute the probability that these sequences of words occur in the language. Chelba and Jelinek (Chelba and Jelinek, 1999) showed that using syntactic-like structures to compute these probabilities leads to significant improvement over simpler methods.

All these applications need good parsers. Improving the parsing performance would also benefit to the applications that need them. Particularly, one of the motivations of this work, which aims at improving the performance of parsing, was to develop a better semantic role labeler, which used parsing.

1.1.1 Modern Parsers

We first need a measure of how good a parser is. A simple automatic way of evaluating the performance of a parser is to compare its results with the correct answers. To perform this evaluation, we need a set of correct parse trees and a distance measure between these trees and the parser trees. Correct trees come usually from a manually parsed corpus called the *gold standard*. The gold standard is a reference to which the parsers are compared. The closer a parser is to the gold standard, the better it is.

To know how close a parser output is to the gold standard, we need a distance measure. A simple measure consists in comparing pairs of trees, one from the parser and the corresponding one from the gold standard, and in counting the number of exactly identical trees. There is a finer measure which is widely used for the evaluation of parsers. It is part of an evaluation scheme called the *Parseval* standard.

The Parseval standard compares subparts of a tree: the *constituents*. A constituent is a subtree of a syntactic tree. In figures 1.1, there are 10 constituents. Their labels are *S, VP, NP, NP, PP, N, V, N, P* and *N* (The letters correspond to the labels *Sentence, Verb Phrase, Noun Phrase, Prepositional Phrase, Noun, Verb* and *Preposition*). But finding the part-of speech tags is not the task of the parser, thus, they are not taken into account for calculating performances. So this tree would be compared to the gold standard using only the five first nodes, which are not part-of-speech tags.

Parseval counts the number of constituents identical to the gold standard. This number is then divided to get the proportion of correct constituents. We can divide the number of correct constituents by two different values: the number of constituents in the parser output, or the number of constituents in the gold. Here is the equation for *precision*:

$$\text{precision} = \frac{|\{\text{constituents in the gold}\} \cap \{\text{constituents in the parse}\}|}{|\{\text{constituents in the parse}\}|} \tag{1.1}$$

Precision shows the number of correct constituents from all those that the parser proposed. Here is the equation for *recall*:

$$\text{recall} = \frac{|\{\text{constituents in the gold tree}\} \cap \{\text{constituents in the parse}\}|}{|\{\text{constituents in the gold tree}\}|} \tag{1.2}$$

Recall shows the proportion of constituents that the parser found, from the constituents that it should have found. To summarise these two measures,

the parsing community uses a third measure called the *F-Score*, which is the harmonic mean of precision and recall:

$$\text{F-score} = \frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \tag{1.3}$$

Now that we know how to evaluate them, we can discuss the performances of modern parsers. They can achieve performances around 88%-90% F-score. These are a high performances, and thus it is difficult to increase them much. An increase in F-score of, for example, 1% is equivalent to an error reduction rate of 10%.

Most errors are due to ambiguity. Ambiguity often arises from the problem of *PP attachment*. A PP is a prepositional phrase like *with Charles* in sentence 1, or *for the difference engine*, in

3. Ada Byron worked on algorithms for the difference engine.

It can be difficult to find what the prepositional phrase modifies. Does *for the difference engine* modify the noun *algorithms*, implying that the algorithms are specific to the engine, or does it modify the verb *worked*, implying that the work specifically benefits the engine ? Church and Patil (1982) noted that “these syntactic ambiguities grow "combinatorially" with the number of prepositional phrases.”

To manage ambiguity, some parsers use a probability associated with each tree, and can output the most probable tree. They need to be trained on a correctly parsed corpus. They learn probabilities from the correct examples of the corpus, and can then use these probabilities to parse new sentences. These parsers are called *statistical parsers*.

1.1.2 Parser Combination

We will now discuss a new technique to increase performances of modern parsers. As we have seen, parsing is a both useful and difficult task. A recent approach to increasing the performances of parsers is the *ensemble learning* framework. Ensemble learning consists in training different classifiers and combining them to increase performance. The method is useful if the performance of the combined classifier is better than the performance of its individual members. A necessary and sufficient condition for this improvement is that the individual classifiers must be *accurate* and *diverse*.

A classifier is accurate if it makes fewer than 50% mistakes. Two classifiers are diverse if they make different errors on the same data. There are several ways to make different classifiers for the same problem. One could, for example, train the classifiers on different data or use different features to train them. More on ensemble learning can be found in (Dietterich, 2000).

Parsers are similar to classifiers. They classify into two categories all the possible constituents that could be part of the tree of a sentence. The categories are: *must be included in the output tree,* and *must not be included in the output tree.* Parsers only output the constituents that they classify in the first category, so any constituent that the parser does not output can be considered as classified in the second category. This suggests that the previous condition for ensemble learning applies also on parsers. Modern statistical parsers are accurate, so, if they are sufficiently diverse, they could be combined to increase performance.

J. C. Henderson and Brill (1999) and Sagae and Lavie (2006), showed empirically that parsers can be successfully combined. They recombined the output of several parsers, and, as a result, increased the global performance. This shows that their parsers were sufficiently diverse to benefit from ensemble learning. They recombined the parsers using voting schemes: for each constituent, the parsers voted on its inclusion in the final tree. They recombined the constituents that were voted by a majority of parsers. This allowed the parsers to recover from errors that only few of the original parsers made. The resulting parses improved the state-of-the-art in both experiments.

These results inspired us to explore more possibilities for parser combination. In this thesis, we will try to improve on previous experiments with several methods for parser combination. In the next sections, we will discuss the differences between previous work and our experiments. In section 1.1.3, we will discuss the diversity of our parsers. Then, in section 1.1.4, we will present a new method to combine the trees. Finally, in section 1.1.5, we will introduce a technique to differentiate correct answers from errors in the parsers output.

1.1.3 Our three Parsers and their Differences.

We will use three different parsers, and recombine their output. Our three parsers all use the same kind of technology, but the nature of their output is different from one another. We will describe the three parsers more in chapter 3.

All parsers are based on the simple synchrony network (SSN) parser by J. B. Henderson (2003). It is a state-of-the-art statistical parser. It builds trees by a sequence of *actions*. To decide which action to take next, the parser takes into account all the previous actions as well as the input sentence.

The other two parsers use the same learning algorithm, but were trained on trees with richer output. The trees were decorated with *function* labels, and *semantic role* labels. The original parser was trained on a corpus with syntactic labels such as *Noun Phrase* or *Verb Phrase*. The second parser (Merlo and Musillo, 2005) was trained on a corpus that indicated not only these labels but also functions tags which give more information about the syntactic role of the constituents in the sentence. The third parser (Musillo and Merlo, 2006a) was trained on a corpus that indicated the syntactic label as well as semantic role labels which give more information about the semantic function of the constituents in the sentence. We will describe these labels further in chapter 3.

All parsers achieved state-of-the-art performances on the purely syntactic task, altough the function and the semantic role label parsers can give a richer output.

Combination of the three Parsers

Each parser uses a different training corpus. The outputs of each parser are different from one another in two ways. First they differ in the kind of labels associated with the nodes: pure syntactic labels, function and syntactic labels, or semantic roles and syntactic labels, respectively. Second, the trees with only the syntactic labels can be different from one parser to another, because the information carried with the semantic role labels and the function labels can be used in the training of the parsers, and affect their behaviour. Table 1.1 shows a summary of the three parsers.

The three parsers based on the SSN parser are all close to the state-of-the-art on syntactic parsing. Since their syntactic outputs can be different, we can assume that they make different mistakes. It should then be possible to combine them to improve performance. This is what we will do in this thesis.

1.1.4 A New Level of Combination

We will now see how we can apply ensemble learning techniques to recombine our three parsers. There are a number of possibilities to recombine

Table 1.1: Summary of the parsers that we will combine.

Parser	Author	syntactic labels	function labels	semantic role labels	F-score on purely syntactic trees
SSN	Henderson	yes	no	no	87.8%
Function parser	Musillo and Merlo	yes	yes	no	88.3%
SRL parser	Musillo and Merlo	yes	no	yes	87.9%

the output of several parsers. J. C. Henderson and Brill (1999) tried two different approaches. The first one is the simplest. For each sentence, their programme chooses the tree that is the most similar to all the other trees. This tree, which is close to the centre of all the observed parse trees, is the final tree. The second approach relies on the decomposition of the trees into constituents. The final tree is then built out of several constituents. These constituents are the ones that were in a majority of observed parse trees. The second method gave better results.

Since smaller subparts give better results, it is interesting to experiment with even smaller parts, a method which, as far as we know, has never been tried. We will decompose the trees into entities that will be smaller than the constituents. We will see that our entities are a better representation of the parser actions than the constituents. This makes them a natural decomposition of the trees.

1.1.5 A New Method for the Classification of Constituents

Parser combination involves recombining correct subparts of trees, in order to get better final trees. This solution relies on the elimination of *false positives*. In the case of parse constituents, a false positive is a constituent that is in the parse tree, but not in the correct tree. A crucial part of parser combination is the discrimination between correct entities and false positives. J. C. Henderson and Brill (1999) and Sagae and Lavie (2006) used systems based on votes from the parsers: a parser voted for a constituent if this constituent appeared in this parser output. Let's consider the parse trees as set of constituents. The parsers are accurate, so there are generally more true positives constituents than false positives in the parse trees. If the parsers make different mistakes from one another, the false positives might be dif-

ferent constituents in each parse tree. Only false positives that are voted by a majority of parsers stay in the final tree. If the parsers are diverse enough, this number will be small.

We will experiment with several systems. We will use votes, but we will also use an automatic classification algorithm. Specifically we will consider the problem of discriminating false positives as a classification problem with two classes: *true positives* and *false positives*. For this task, we will use a *support vector machine*. Support vector machines can learn to classify entities using a number of clues. They learn the importance of each clue, and can ignore useless ones. We will feed our support vector machine with several clues. One of them will be the vote of each parser, but we will also use other clues, such as the label of a constituent, or the length of the sentence.

1.2 Outline of the Thesis

This thesis is organised as follows. In chapter 2 we will summarise previous works on parser combination, by other authors. In chapter 3, we will describe the three parsers that we recombined. In chapter 4 we will discuss the questions that motivated us to do this work, and the hypotheses that we made. In chapter 5 we will describe our first experiments and their results. After the firsts experiments we increased the complexity of the system and tried further experiments, which we will describe in chapter 6. Finnally, in chapter 7, we will give some conclusions.

1.3 Conclusions

We base our research on the work of (Henderson and Brill, 1999) and (Sagae and Lavie, 2006), which proved that it is possible to achieve better performance by combining several parsers. We will do more experiments with the three following parameters. We will use parsers which are different from one another, not in the kind of technology they use, but in the richness of labels they output. A necessary condition for ensemble learning is that the errors must be different for each parser. If we get good results, we will thus verify the difference of the parsers due to the different label sets.

We will, also, decompose the trees into subparts of different sizes. These experiments will show which is the optimal level of decomposition.

Finally, we will investigate several methods to discriminate the correct subparts of trees from the false positives. We will use, among others, a machine learning algorithm to select the relevant features for this classification.

Chapter 2

Related Work

We will now present some previous works by other authors on the combination of parsers (2.1) and on support vector machines, which we will also use to discriminate correct entities from false positives (2.2).

2.1 Parser Recombination

The work of this thesis was much inspired by the work of J. C. Henderson and Brill (1999) and by the work of Sagae and Lavie (2006). Both achieved an improvement of the parsing performances by combining the output of several statistical parsers.

2.1.1 Previous Work by J. C. Henderson and E. Brill

J. C. Henderson and Brill (1999) used ensemble learning methods to improve the results of three different state-of-the-art statisticall parsers. They used four different techniques to combine the output of the parsers.

In the first technique—parser switching—the final parses were entire trees, each one from one of the parsers outputs. For each sentence, J. C. Henderson and Brill choose the most consensual tree out of the trees from their parsers output. Formally, the most consensual tree is the tree that has the most common constituents with the other trees.

The second technique—constituent voting—was to decompose the trees into constituents and to form a new tree out of the most probable of those constituents. J. C. Henderson and Brill made the parsers vote for the inclusion of each constituent, and they created a set out of all the constituents

that were in the output of a majority of parsers. This set of constituents forms a new tree. Not any set of constituents forms a tree because the constituents can have crossing parentheses, that is two constituents can include the same word, without them being embedded in each other. However, J. C. Henderson and Brill showed that the set of majority constituents cannot contain constituents with crossing parenthesis, and that it forms a tree.

J. C. Henderson and Brill also developed two methods that are similar to the previous ones, but with a more complex model of the probabilities that a constituent is a true positive, given the votes of the parsers. They used a probabilistic Naive Bayes model, which assumes that the parsers make independent prediction for each constituent. In this model, the probability of a constituent can be calculated using the probability that any constituent proposed by a parser is a true positive and the probability that a given parser proposes a constituent if the constituent is a true positive. Those parameters can be calculated from a parsed training corpus.

J. C. Henderson and Brill adapted the method where the trees are decomposed into constituents to use the Naive Bayes model. The new set of constituents that forms the final parse is then the set of constituents that have a probability greater than 0.5 of being a true positive. In the parser switching version, they calculated the probability of the entire trees by multiplying the probabilities of their constituents, and they choose the most probable tree for each sentence.

J. C. Henderson and Brill also considered giving preference to certain parsers that are better than the others in some context. That is, for the constituent voting method, they would sometimes include a constituent that was voted by only one parser, because the solitary parser would be better than the other two under the right circumstances. They calculated the possible improvement of this technique when taking into account the performances of the parsers for each constituent label. They calculated, for each label, how many time a parser was right in including the constituent, when the other two were wrong. They found that the constituents voted by only one parser were mostly false positives, and the few labels for which a parser alone gave more true positives than false positives included very few constituents in the corpus that they studied. Therefore, using labels as contextual information was useless.

All methods, with trees or constituents and with or without Naive Bayes models, were successful and the Naive Bayes model led to improvement for the method with entire trees. Constituent voting achieved a better F-score and precision than parser switching, but the recall degraded. An F-score of

90.4% was achieved for the simple parser switching experiment. The Bayes parser switching method led to a 90.7% F-score and both the experiment with constituents gave 91.3% F-score.

The work of J. C. Henderson and Brill shows that ensemble learning is applicable to the parsing problem. It shows that improvement can be made by choosing the most consensual entire tree as well as by choosing the most voted constituents. Our work is very similar. We also recombined the output of several statistical parsers. Part of our work was to ensure that even parsers that might be less diverse than the ones that J. C. Henderson and Brill used can be successfully combined, when using the same constituent voting method.

The fact that combining smaller entities—constituents—gave better results than combining bigger entities—trees—inspired us to decompose the constituents into even smaller entities: parentheses. We will give more motivation and detail for this decomposition in section 4.2

We did not, however implement a probabilistic model in our work, as J. C. Henderson and Brill did not achieve better results with their probabilistic model than with the simple method, when they decomposed trees into constituents. We experimented with the same voting scheme on constituents as J. C. Henderson and Brill did, but we also experimented with other methods for choosing the best constituents. We gave the constituents a weight that was proportional to the votes, but we did not discard all minority constituents as J. C. Henderson and Brill did. The reason was that the voting scheme tends to favour precision over recall, as we will see in the next section. Moreover, we could not use the voting scheme for our smaller entities because it was not directly applicable, and we had to develop new schemes to handle them.

2.1.2 Previous Work by Sagae and Lavie

Sagae and Lavie (2006) also combined the output of several parsers to get better performances than the individual parsers. They performed two experiments, one with dependency parsers and one with constituent parsers. We will now summarise the latter.

They used five state-of-the-art constituent parsers. They decomposed their output into a set of constituents. Each constituent was associated with a weight proportional to the number of parsers that output the constituent. Constituents whose weight was lower than a threshold were discarded. Fi-

nally an algorithm that output the heaviest tree out of the remaining constituents was used. This tree was the final parse.

If the weight of each constituent is the votes of the parsers and the threshold is $\frac{m+1}{2}$ where m is the total number of parsers, then the experiment is very similar to the work of J. C. Henderson and Brill (1999). However, this scheme tends to favour precision over recall.

Sagae and Lavie used two ways of improving their basic experiment. First they calculated the precision of each individual parser for each label of constituents. Then they calculated the weights of the constituents by multiplying the votes by the precisions of the parsers for the label of the constituents, so that the weight was proportional to the number of votes and the certainty of the votes. Formally, if $P(l, i)$ returns the precision of the i^{th} parser for the constituent bearing the label l, and if $V(c, i)$ returns 1 if the i^{th} parser included the constituent c in its output, then they used $\text{weight}(c) = \sum_i P(l, i)V(c, i)$ where l is the label of c.

Second, they used a learning algorithm to find the best possible threshold. They used a training corpus to automatically try different threshold and select the best. They optimised the threshold to suit two different needs: they found the threshold that minimises the difference between precision and recall, and the threshold that maximises the F-score.

They obtained a 91.6% F-score for the $\frac{(m+1)}{2}$ threshold, 91.8% for the threshold that minimises the difference between precision and recall (91.8% and 91.9%, respectively), and 92.8% for the threshold that maximises the F-score. The F-score of each experiment was better than the F-score of the best of the individual parser.

The work of Sagae and Lavie inspired us not to discard all the minority constituents since they showed that this scheme favours precision over recall. When the set of constituents contains minority constituents it might contain constituents with crossing parentheses and thus the set might not form a tree. Since we wanted to output a tree, we had to develop an algorithm to build a tree out of the heaviest constituents. We developed a new algorithm based on a well known parsing algorithm, that is able to find the best solution in a $O(n^3)$ time. This algorithm was also useful when we recombined parentheses, because then the set included constituents that were not in any of the parser output.

Similarly to the work of Sagae and Lavie, we experimented with different

thresholds, however we did not use a learning algorithm to find the best threshold. Since we had only three parsers, we had few different possible weights, and we could experiment with each possible threshold manually.

2.2 Support Vector Machines

An important part of our work is to discriminate, among the constituents proposed by the parsers, the true positives from the false positives. We will perform experiments for this classification with an automatic classification algorithm: the support vector machine. We will now give a brief introduction to this algorithm.

A classification problem can formally be represented in the following way. Each item to be classified is represented as a vector of features. The features are attributes of the item, that are tought to be useful clues to guess the class of the item. In binary classification, the class of the item is a binary value.

The goal of a support vector machine is to find a function of the feature vectors that returns 1 if the item belongs to the first class and -1 otherwise. The algorithm learns such a function by looking at examples with the correct answers in a training corpus. It adapts the function to the examples, and can then use this learnt function to classify new items with unknown answers.

Formally, the function defines an hyperplane that separates the input vectors into two classes. It can happen that it is not possible to separate the two classes with an hyperplane: the problem may not be linear. In this case it is possible to map the vectors into an higher dimensional space where they might be linearly separable. A *kernel* defines the function that does this mapping, and several different kernels can be used (Fradkin and Muchnik, 2006).

Hsu, Chang and Lin (2003) developed a methodology and a programme that allows us to obtain acceptable results fast and easily with a support vector machine. They proposed to use the *radial basis function (RBF)* kernel, because it can handle several different cases, including the linear case, and because it uses reasonable processing time. The support vector machines can be tuned with some parameters, and Hsu Chang an Lin's programme also learns the best parameters.

Hsu , Chang and Lin showed that their approach gave good results on a range of classification problems, if the feature set was relatively small (about 20 features).

2.3 Conclusion

The parser combination problem is not a new one. We presented the work of J. C. Henderson and Brill and of Sagae and Lavie, which already showed that it is possible to improve the parsing performances by combining the output of parsers following the ensemble learning framework.

Our work relies much on these previous works but it also introduces new experiments. One new experiment in our work is the decomposition of constituents into smaller entities. Those entities necessitated a new weighting scheme and the development of a tree building algorithm.

We also introduced the support vector machine, which is a powerful classification algorithm. We will use this algorithm and the programme developed by Hsu, Chang and Lin to discriminate the false positives from the true positives among our constituents.

Chapter 3

The Three Parsers

The goal of this thesis is to improve on the performance of three parsers, by recombining their output. In this chapter we will describe in more detail the resources that we will use: the gold corpora and the three parsers.

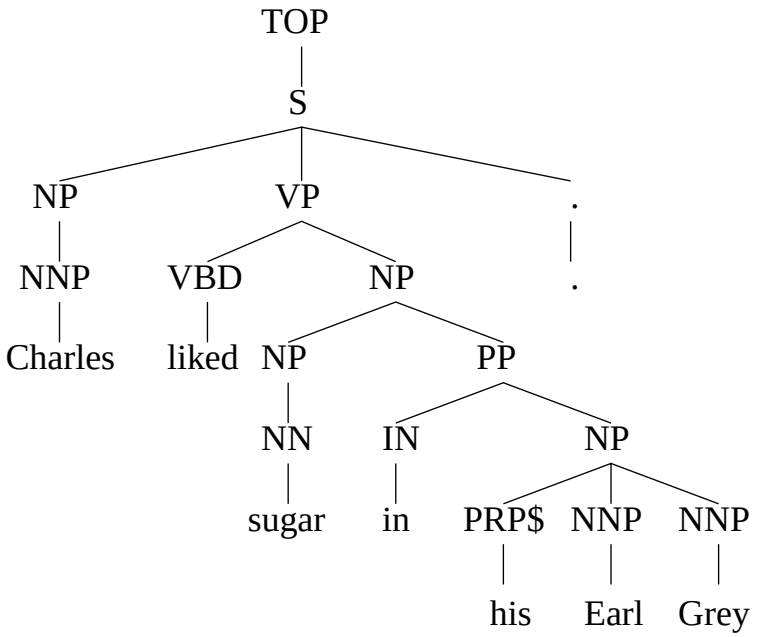
3.1 The Penn Treebank

A set of correctly parsed sentences is called a *treebank*. A treebank is useful for the automatic evaluation and the training of statisticall parsers. In an automatic evaluation scheme such as Parseval, a treebank can be used as a *gold standard*: a set of correct answers to the problem at hand. Parsers that are based on supervised machine-learning algorithms also need a treebank. They use a set of correct parse trees as examples, from which they can learn. They generalise from these examples and learn to parse new sentences.

The Penn treebank (Marcus et al., 1994) is a large treebank. It consists in a manually annotated corpus. The annotators constructed the parse trees by correcting the output of a parser. There can be some annotator mistakes, but this problem is inherent to manual annotation. There are several level of annotations in the Penn treebank: the part of speech tags, the skeletal syntactic structures, and the predicate argument structure.

3.1.1 Part of Speech Tags

The part-of-speech tags are the lexical categories of the words: nouns, adjectives, verbs, etc. The Penn treebank uses 36 different part-of-speech tags, and 12 others for punctuation and currency symbols.



(TOP (S (NP (NNP Charles))(VP (VBD liked)(NP (NN sugar)(PP (IN in)(NP (PRP$ his)(NNP Earl)(NNP Grey)))))(. .)))

Figure 3.1: A tree and a parentheses representation of a parsed sentence

Here is an example of a sentence with part of speech tags:

1. Charles liked sugar in his Earl Grey .
   NNP VBD NN IN PRP$ NNP NNP .

NNP is the tag for a singular proper noun, VBD for a verb in past tense, NN for a singular or mass noun, IN for a preposition, and PRP for a personal possessive pronoun (Taylor et al., 2003). The part-of-speech tags are the pre-terminal nodes of a parse tree, and the words are the terminal nodes.

3.1.2 Skeletal Syntactic Structures

The trees of the Penn treebank are represented as sentences annotated with parentheses. There is a one to one match between a tree, and a sentence with parentheses. Consider the previous sentence. Figure 3.1 shows the tree associated with it, and the corresponding parentheses representation.

Each pair of parentheses represent a constituent. The innermost ones are the part-of-speech tags. The *TOP* label is an artificial label which always

appears at the root of the tree.

The tree in 3.1 is annotated with skeletal syntactic structures. The constituents of the tree bear a syntactic label. There are 14 possible labels, and 4 null elements. Null elements are used to represent constituents which do not appear in the sentence but are necessary to the syntactic structure. Consider the following example:

2. Who does he want to program ?

It can be argued that the natural order of this sentence is:

He wants who to program ?

Which reflects the order of the assertive sentence:

He wants Ada to program.

It is said that the interrogative *who* is *generated* in the middle of the sentence and is moved to the beginning, leaving an invisible trace in its original position. The sentence would then be:

Who does he want *trace* to program.

This has several consequences, for example, it can be argued that the presence of the trace prevents the contraction of *want* and *to*. Altough the Penn treebank has annotations of null elements such as traces, the parsers do not take them into account, and we will not discuss those annotations any further.

3.1.3 Function Labels

In the Penn treebank, the skeletal syntactic trees are also decorated with richer labels that carry more information about the semantic and grammatical role of the phrases in the sentence. These labels are finer grained than the skeletal labels. So, for example, a NP will also be a subject or an object. There are 17 function tags that can be appended to the skeletal labels. Figure 3.2, from (Musillo and Merlo, 2005), shows a tree annotated with some of the function tags. The NP *the authority* carries the function tag *subject*, and the PP *at midnight* carries a function tag indicating that it is temporal. The tree still carries all the skeletal syntactic structure, but with more information.

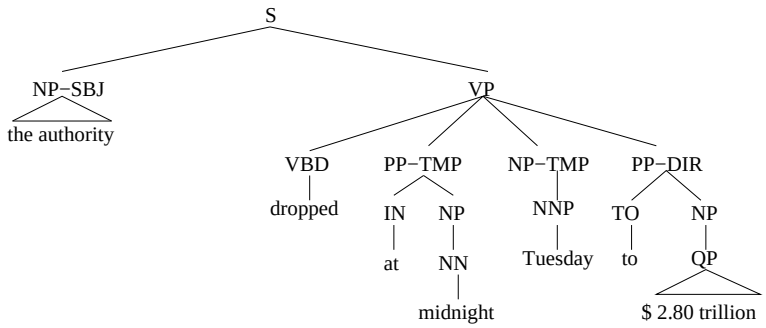


Figure 3.2: A tree annotated with function tags. The tags represent subclasses of each syntactic tag.

3.2 The Prop Bank

The Proposition bank (Palmer et al., 2005) is the Penn treebank with trees annotated with more information than the syntactic structure: semantic labels. Palmer and her colleagues added two layers of semantic annotations to the trees of the Penn treebank. First, they defined a number of arguments for each verb, and numbered them. For example, the verb *to open*, with the sense *to cause to open*, has three arguments: the agent, the thing opened and the instrument (example from (Palmer et al., 2005)). The phrases of each sentence of the Penn treebank were annotated with the number of the argument they correspond to.

Second, the Proposition bank annotates adjuncts to verbs. This annotation is similar to the function label annotation of the Penn Treebank and uses labels such as *location time* or *cause*. These annotations are the same for every verb in the treebank (Musillo and Merlo, 2006b).

Figure 3.3, from (Musillo and Merlo, 2006b), shows a tree of the Proposition bank. There are three arguments to the verb *drop*: A1, A4 and A3. There are two adjuncts, both temporal: *at midnight* and *Tuesday*. The original syntactic structure from the Penn treebank is still present, but the semantic information is added to the syntactic labels, making them finer-grained.

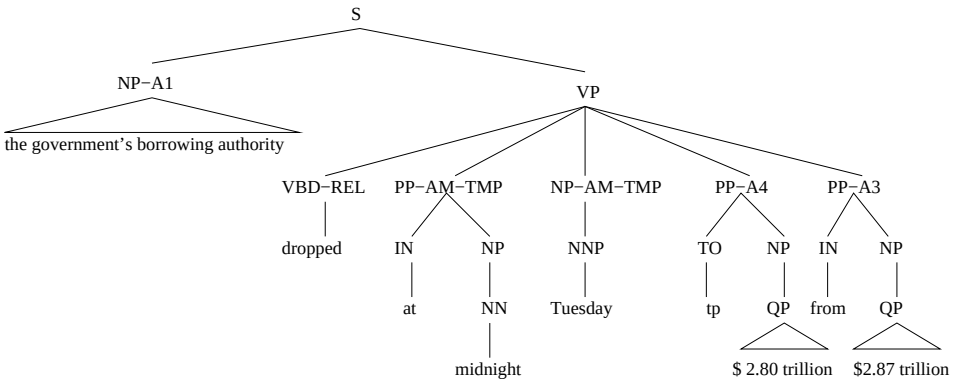


Figure 3.3: A tree annotated with semantic role labels. There are three arguments to the verb and two adjuncts.

3.3 The SSN Parser

We will now describe the first parser that we used in our recombination. It is the Simple Synchrony Network (SSN) parser described in (Henderson, 2003).

The SSN parser is a statistical parser: it uses a machine learning approach. It learns a probabilistic model of parsing by looking at correct parsing examples from a training corpus. Many statistical parsers represent trees as a sequence of *actions*. These actions are moves required to build the tree, such as creating or attaching a node. These statistical parsers can then output the most probable tree which is the tree composed of by the most probable actions at each step. The parsers can use an *history-based* approach: they calculate which is the most probable correct next action given the input sentence and all the previous actions that they have performed to produce the tree so far. The challenge with this approach is that the number of relevant previous actions at any moment is not bounded. Many parsers solve this by using only a fixed number of defined features of the history. They make an independence assumption: they assume that the probability of the next action is independent of the other features.

The SSN parser does not make this assumption. J. B. Henderson uses a specifically designed neural network (the simple synchrony network (Lane and Henderson, 2001)) to calculate a finite representation of the history at each step. Since this representation is calculated recursively, taking into account each time the previous representation, there is no independence assumption.

J. B. Henderson's parser consists in two main levels. The first one is

a neural network that calculates the history representation, and the second one is the parsing algorithm that computes the most probable next action given the history representation and the input sentence. Neural networks are algorithms that allow a programme to learn from correct examples, generalise from them, and give the correct answer to new examples. The parser must, therefore, be trained on a correctly parsed corpus before it can parse new sentences.

The actions that the parser can take to construct the trees are modifications of a stack and of the parse tree. At the beginning the stack only contains the label *root*. The actions are (Henderson, 2003):

- Shift(w): puts the current tag-word pair w on top of the stack.
- Project(y): removes the token X of the top of the stack and replace it with the token Y. It also specifies, in the tree, that Y is the parent of X.
- Attach: removes the token X from the top of the stack. Specifies that the next token Y of the stack is the parent of X in the tree.
- Modify: removes the token X from the stack. Specifies that the next token Y of the stack is the modifier parent of X in the tree.

The neural network that calculates the representation of the history receives as input the last representation of the history as well as some defined features of the history. Choosing the features that are directly passed to the neural network is a way to implement *soft biases*. Soft biases are ways to give more importance to some features of the history that can be used to encode *a priori* linguistic knowledge into the parser. J. B. Henderson got improvement in his results when he used carefully chosen soft biases. His SSN parser achieved state-of-the-art results: 89.1% F-score on the Penn Treebank.

3.4 The Function Parser

The original SSN parser outputs purely syntactic trees. However, as we have seen in section 3.1.3, the trees of the Penn Treebank also carry richer function labels. Musillo and Merlo (Musillo and Merlo, 2005) trained the SSN parser on those labels. We will now describe the resulting parser: the *function parser*.

As Musillo and Merlo (2005) noted, before their work the standard approach to getting function tags was to use a two level system were a sentence

was first parsed then decorated with those labels. Musillo and Merlo, however, developed a system in which both tasks are performed together. They retrained the SSN parser on a corpus annotated with the function tags. The structure of the trees were the same, but all labels were finer grained (188 labels instead of the original 33). They used biases to give more importance to the part of the history relating to nodes that are syntactically close to the node that the parser is guessing. Their parser not only gave state-of-the-art results on the function parsing task, but also the performance did not decrease compared to the SSN parser on the purely syntactic task, altough the task was more complex and the output was richer. The parser could take advantage of the richer information to better learn the original task.

The function parser achieves 89.2% F-score on the Penn Treebank for the purely syntactic task.

3.5 The Semantic Role Labelling (SRL) Parser

Musillo and Merlo (Musillo and Merlo, 2006a) trained another parser in a similar fashion to the function parser. This time they trained the SSN parser on the Proposition bank (see 3.2). They used a tagset of 613 non-terminals instead of the 33 purely syntactic tag that the SSN parser uses. The new tags are made of the original tag set followed by one or more semantic roles from the Proposition bank.

Their new parser had to solve a more complex task than the original parser since it not only output the syntactic structure but also the semantic role labels. It gave satisfactory results on the semantic role labelling task, and the performance on the purely syntactic task did not significantly decrease (89.0% F-score on the Penn treebank), altough the task was more complex.

3.6 Conclusion

In this chapter we described a corpus with syntactic structure information: the Penn treebank. We saw that the Penn treebank had also been annotated with semantic role labels resulting in the Proposition bank.

In this work we will use three parsers: the SSN parser, the function parser and the SRL parser. We will combine their results to improve the performance of parsing. One of the novelty of this work is that the three

parsers are not different in the technology that they use, but in the granularity of the tags on which they were trained and that they can output. The technology used for all three parser is the same: It is based on the parser developed by (Henderson, 2003). As we will see in the next chapter, one of the question underlying this work is whether the parsers are diverse enough to be successfully combined.

Chapter 4

Research Question: How can we get a Better Score with our Parsers ?

The goal of this thesis is to combine three parsers that are all based on the SSN technology, but that were trained on corpora whose label inventory had a different granularity and conveyed different types of linguistic information. This combination will be successful if it increases the performance, compared to the performances of the individual parsers. In this chapter we will examine in detail the research questions that underlie this work, as well as the hypotheses that we will make.

4.1 Can we Successfully Apply Ensemble Learning to our Parsers ?

We will consider that the combination of several programs is successful, if the resulting performance is higher than the performance of the individual programs. As we have already seen in chapter 1, Dietterich (2000) gives the following condition for the combination of several classifiers: the combination will be successful if the classifiers are accurate and diverse. We know, from previous work such as (Henderson and Brill, 1999) and (Sagae and Lavie, 2006), that it is possible to successfully combine the output of several parsers. We will investigate if this result applies to our SSN parsers, that is, if our parsers satisfy the condition. Our parsers are accurate: they have a precision and a recall higher than 50%. The question is whether they are diverse enough.

The parsers are diverse if their errors on new inputs are independent from each other. An error that is made by all parsers will also occur in the final parse, whereas it is possible to detect and correct an error that is made by few parsers. This means that the more correlated the errors are among the parsers, the less successful the combination will be.

It is not straightforward to determine if our parsers are diverse, because, contrary to previous work, we use parsers that are implemented with the same technology, and this could make the errors more correlated. There are several reasons to believe that this will not be an obstacle. We will examine them in details in the next sections. In section 4.1.1, we will calculate how the performance would increase if the errors were independent. In section 4.1.2 we will present a minimal condition on the correlation of errors, for the combination to be successful. We will then present an upper bound on performances for our specific data.

4.1.1 Upper Bound with Independent Errors

The more independent the errors, the better the combination will be. We will now calculate an upper bound for the combination, if it is done with a voting technique, and if the errors are totally independent between parsers. If this upper bound were too low, our work would not be relevant, but we will see that it is high.

The voting technique consists in choosing only constituents that are in a majority of observed trees, so that only errors that appear in two or three of the parsers would also appear in the final trees. We calculate the upper bound of precision first. The original parser achieves a precision of 88.4%, the function parser 88.9% and the SRL parser 88.3%, that is, error rates of 11.6%, 11.1% and 11.7% respectively. If the errors of the parsers are independant, then we can multiply their probabilities to obtain the probability that two or three parsers make an error. The probability that only the original and the function parsers make an error is $0.116 \cdot 0.111 \cdot 0.883$. The probability that only the function and the SRL parsers make an error is $0.884 \cdot 0.111 \cdot 0.117$. The probability that only the original and the SRL parsers make an error is $0.116 \cdot 0.889 \cdot 0.117$. The probability that all parsers make an error is $0.116 \cdot 0.111 \cdot 0.117$. So we have:

$$\begin{aligned} &0.116 \cdot 0.111 \cdot 0.883 \; + \; 0.884 \cdot 0.111 \cdot 0.117 \\ &+ 0.116 \cdot 0.889 \cdot 0.117 \; + \; 0.116 \cdot 0.111 \cdot 0.117 = 0.036 \end{aligned}$$

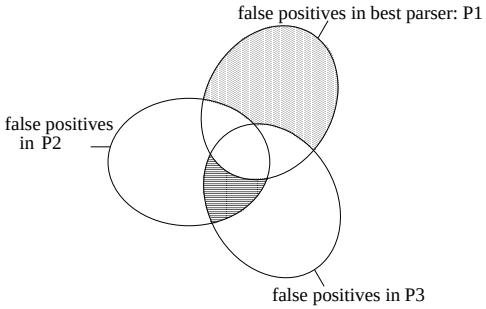


Figure 4.1: False positives in each parser. If the cardinality of the upper grey area is lower than the cardinality of the lower grey area, the combination will be successful.

So, if the errors are independent, the false positive rate in the final parse is 3.6%, which represent a precision of 96.4%. The same reasoning applies to recall:

$$\begin{aligned} &0.127 \cdot 0.123 \cdot 0.876 \ + \ 0.127 \cdot 0.877 \cdot 0.124 \\ &+ \ 0.873 \cdot 0.123 \cdot 0.124 \ + \ 0.127 \cdot 0.123 \cdot 0.124 = 0.0427 \end{aligned}$$

So the upper bound recall for voting is 95.7%.

Unfortunately, we cannot assume that the errors are independent. The parsers may encounter similar difficulties, and some constituents may be more difficult than others for all parsers, so the errors are probably not totally independent. But improvement can be made if the errors are not highly correlated. We will now present a minimal condition on the distribution of errors for the combination to be successful.

4.1.2 A Minimal Condition on the Correlation of Errors

Let us formalise what the precision of a successful combination is. The same reasoning applies to recall, but applied to false negatives. Precision increases if there are fewer false positives in the final parse than in the best parser output.

Let P_1 be the parser that produces the least false positives, and P_2 and P_3 the other two parsers. If we use votes, we keep all the false positives that are in the intersections of two or three parsers. Figure 4.1 shows the set of false positives. The combination is successful if there are fewer false positives

in the intersections of the 3 sets than in P_1:

$$\begin{aligned} \mid P_1 \cap P_2 \mid + \mid P_1 \cap P_3 \mid + \mid P_2 \cap P_3 \mid -2 \mid P_1 \cap P_2 \cap P_3 \mid \\ < \mid P_1 \mid \end{aligned}$$

The intersections involving P_1 are shared and can be subtracted from both sides:

$$\begin{aligned} \mid P_2 \cap P_3 \mid - \mid P_1 \cap P_2 \cap P_3 \mid \\ < \mid P_1 \mid - \mid P_1 \cap P_2 \mid - \mid P_1 \cap P_3 \mid + \mid P_1 \cap P_2 \cap P_3 \mid \end{aligned}$$

This means that the cardinality of the lower grey area in figure 4.1 must be smaller than the cardinality of the upper grey area. P_1 must more often be the only parser which is mistaken than the only parser which is right, for the parse combination to bring an improvement in principle.

We counted the cardinality of these sets in our parsers output for section 24 of the Penn Treebank, to verify that the condition applies. The best parser is the function parser. The results are given in table 4.1.

Table 4.1: Some analysis of the results of the parsers to calculate the upper bound for ensemble learning.

	Only in the function parser	In both other parsers
false positives	1087	406
false negatives	650	426

The cells on the left contain higher numbers than the cells on the right, so the condition is verified for our parsers, and the combination could be successful. Experiments will show how much.

We verified that the increase in performance can be significant by calculating a more precise upper bound. We used the output of our parsers on section 24 of the Penn Treebank. We calculated this upper bound for two different combination techniques: recombining constituents, and recombining smaller entities. We wanted to verify the presence of correct constituents in at least one parser output, and the absence of some false positives in at least one parser output. The exact procedure is described in section 5.5.

The upper bound performances were much higher than the best parser performances and than the state-of-the-art: 93.9% maximum recall and 96.3% maximum precision.

4.2 How do we Decompose the Parses before Recombining them ?

As we explained in chapter 2, previous works use mainly two levels of decomposition: one consists in selecting the best whole trees out of the observed parse trees, and the other consists in using the best constituents in the observed parse trees to build new trees. The second approach gave better results. Conceptually, a bigger entity might contain more mistakes than a smaller correct entity. For example, if we choose the most correct tree out of several trees that all count several false positives, we will still have false positives in the best possible output. Those mistakes could be corrected by choosing individually the correct constituents, because the correct constituents might be in other trees. The most correct bigger entity cannot contain less error than the entity made out of the most correct sub-entities. At the limit, if the entities are atomic, choosing always the correct ones produces a perfect result. We will, thus, examine if we can further decompose the constituents.

A *constituent* is defined as a triplet: (label, opening position, closing position). *Label* is the syntactic label of the constituent, *opening* and *closing position* are the indices of the first and of the last word of the constituent. For example the tree in figure 4.2 includes the following constituents: (S, 1, 8), (NP, 1, 2), (VP, 3, 8), (VP, 4, 8), (PP, 5, 8).

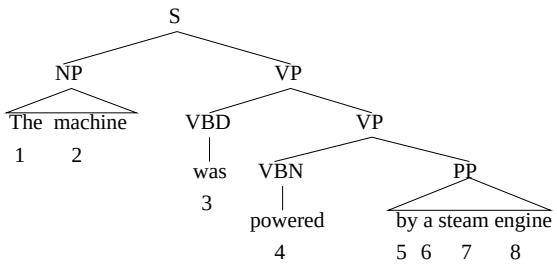


Figure 4.2: A syntactic tree with word indices. The tree includes several constituents. Two of them bear the *VP* label.

As we have seen in chapter 3, the parsers build the trees using sequences of actions. The actions never involve actual constituents, but they can be seen as manipulating parts of constituents. Specifically, the project(Y) action decides on the opening position of a constituent, and some attach actions decide on the closing position of a constituent. Project(Y) introduces a new constituent Y, and specifies that the first child of Y is X: the constituent on the top of the stack. This specifies that the opening position of Y is the opening position of X. Similarly, the attach action that attaches the last child of a constituent, defines this same constituent closing position. Although the match between actions and opening and closing positions of constituents is not bijective, the opening and closing positions are more similar to the parser actions than the constituents.

The fact that they are closer to the parser actions than constituents makes the opening and closing positions a natural choice for decomposing the constituents. We will extract two pairs from each constituent: (label, opening position) and (label, closing position). We will call these pairs *parentheses*, because they match the parentheses in the parentheses representation of the trees. We will then recombine these pairs into new constituents. We will select among them the constituents that form the final trees. It is possible that some of these recombined constituents do not appear in any of the observed trees.

We will call a constituent formed of two half constituents and that is not necessarily in any observed tree a *recombined constituent*. In contrast, we will call constituents that were not decomposed *whole constituents*. We will make experiments with recombined constituents and with whole constituents.

4.3 Which are the Correct Entities ?

Independently of the size of the entities that we combine, we need a method to discriminate the true from the false positives. As we have seen in chapters 1 and 2, J. C. Henderson and Brill (1999) had good results with votes. Sagae and Lavie (2006) generalised the technique by using a weighing scheme. They attributed a weight to each constituent, and combined the heaviest constituents into the final trees. They combined only constituents that were heavier than a threshold. They expressed J. C. Henderson and Brill's method in this framework: the weight is the number of votes, and the threshold is the majority of the parsers.

We will give whole constituents a weight based on votes and determine what is the best threshold. However, we need to adapt this method to recombined constituents. We will use two different weighing schemes for recombined constituents. The voting technique cannot be adapted straightforwardly, because, although the case of one whole constituent appearing several times in one parse tree is impossible, the same parenthesis can appear many times in each tree. For example, several NPs can begin at the same position, leading to many opening parenthesis of the same label at the same place. We will, thus, use a weight proportional to the number of observed trees in which each parenthesis occur, and another weight, proportional to the total number of occurrences of each parenthesis in the observed trees. Experiments will show which weighing scheme gives better results. Those weighing schemes will be discussed in detail in chapter 5.

We will also use a support vector machine classifier to classify the constituents into two classes: *true positives* and *false positive*. We hope that using more parameters to discriminate false positives will lead to better results. Here the parameters will be the kind and number of features on which the support vector machine will be trained. Particularly, the experiment will show if the labels of the constituents can help in discriminating false positives or are useless, as they were for J. C. Henderson and Brill (1999).

4.4 How do we Build a Tree out of a Set of Constituents ?

The Parseval standard, which we will use to evaluate our results, applies on sets of constituents. A tree is always a set of constituents, however, the converse is not true: a set of constituents does not necessarily form a tree. For example, if two constituents in the set have crossing parentheses then they cannot coexist in a tree. It is, thus, possible for a set of constituents that is not a tree to have a good score under the Parseval standard. Specifically, it can get a 100% recall, but it cannot get a 100% precision, because the corresponding tree in the gold standard cannot include a pair of crossing constituents.

We want to output trees, because a parse which is not a tree has little syntactic meaning. So we will use an algorithm that builds a tree out of the best constituents. This algorithm will also discard bad constituents that compete with better ones, improving precision. We need an algorithm that outputs the heaviest tree out of a set of weighted constituents.

Such an algorithm does not, as far as we know, readily exist. There exists, however, several algorithms that find the best tree out of a weighted grammar. We will adapt and use the Cocke-Younger-Kasami algorithm, which, being based on dynamic programming, is a fast parsing algorithm.

4.5 Conclusion

Our main objective is to find the best way to combine our three parsers. We will explore several parameters. We will use two different sizes of tree subunits: constituents and parentheses. We will select the whole constituents with two methods: choosing the constituents that get most votes, or using a support vector machine to discriminate them. We will select the recombined constituents with two weighing schemes. The first weighing scheme is based on one vote per parser, and the second is based on n votes per parser. We will explore the space of these parameters to find the best results.

Chapter 5

Development of a Simple System

In this chapter, we will describe our first experiments and present their results. Particularly, in section 5.3, we will examine how we decomposed the parses before recombining them. In section 5.5, we will describe how we adapted an existing algorithm to output the heaviest tree out of a set of constituents. We will then (5.7) present two simple experiments and analyse their results. These results led us to develop a more complex system.

5.1 Global Flowchart

We developed a program to recombine the parses of our three parsers. We used this program with different parameters to test our different hypotheses. Figure 5.1 describes the general structure of this program.

First, each parser processes a corpus of sentences. The resulting parses are stored in files using the parentheses representation (for details about this representation, see section 3.1.2). Our program retrieves those trees. The program processes sets of three corresponding trees, one from each parser, each tree representing the same sentence. It decomposes the trees into constituents, and merges all constituents of all trees into a multiset. This is the set of all possible constituents for this sentence. At this point, we can perform two different experiments: either the program uses the whole constituents, or it decomposes the constituents into parentheses and then recombines them (for more on the recombination see section 5.3). It then gives a weight to the whole or recombined constituents (for more information on this step, see section 5.6). The set of weighted constituents is then passed to a function that selects those that form the heaviest possible tree (see section 5.5). This

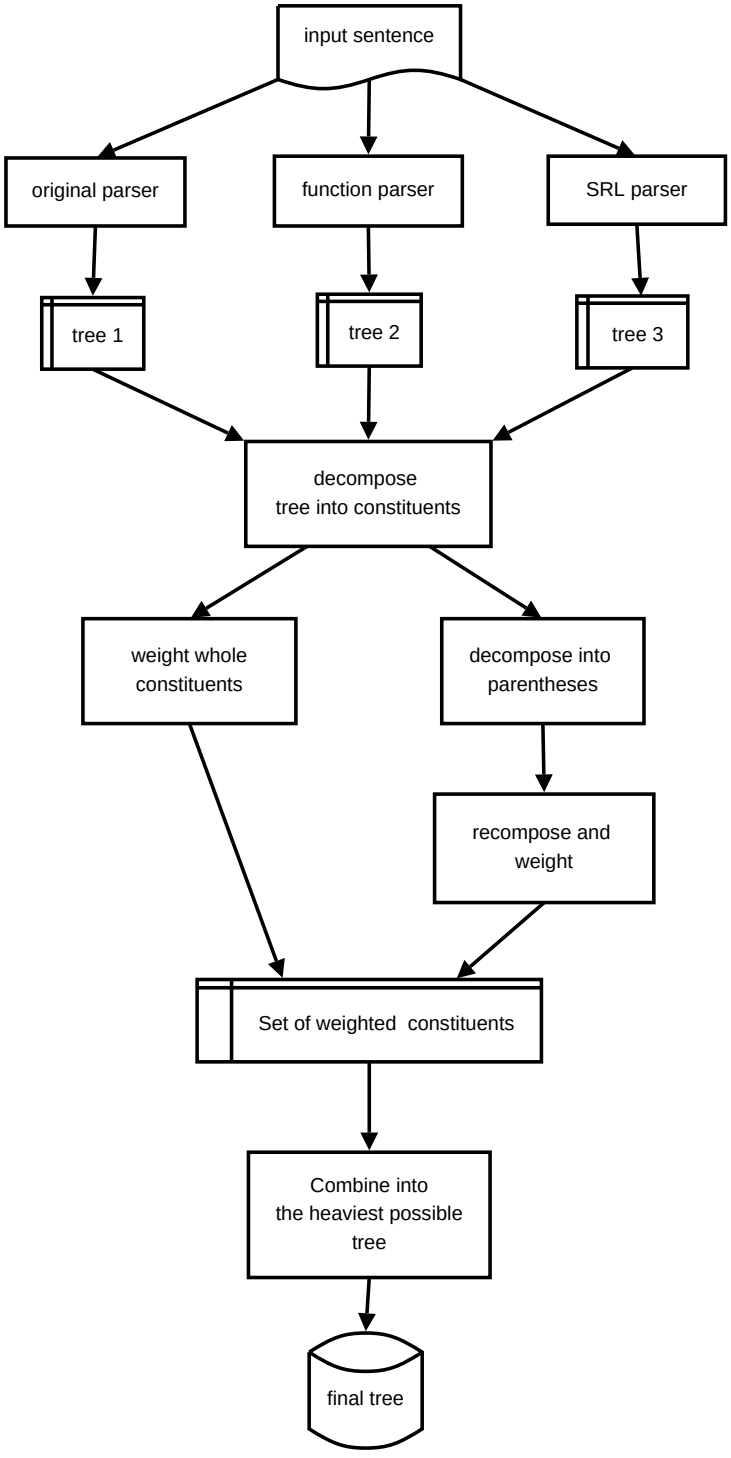


Figure 5.1: Flowchart of the simple system.

tree is then saved into a file. Finally, we can evaluate the quality of the resulting trees using the Parseval measure.

5.2 Data and Materials

We will now briefly present the data and the materials that we used for our experiments.

We performed all experiments on section 24 of the Wall Street Journal part of the Penn Treebank. This is one of the sections that was never used to train our parsers, and, therefore, it constitutes new data. It is a relatively small section, which would not be suitable for training, but having a small corpus helps to reduce the run-time of our experiments. Although the section is small it still gives a significant amount of results. This section contains 1'345 sentences and about 25'600 constituents.

The input of our program was the parses of the sentences of section 24 by the SSN Parser, the function Parser, and the SRL Parser. We never used the SRL or function tags, but only the syntactic tags of the parses. When evaluating our results, we used section 24 of the Penn Treebank as gold standard, and we took into account only the syntactic tags. For the evaluations, we used the program *evalb*, which implements the Parseval standard (Sekine and Collins, 2005).

We ran all experiments on a linux machine, with two Intel Pentium 4 3.2GHz processors and 1 Gigabyte of RAM. We wrote the programs in Perl, using the package *Tree::Binary* for the trees. The run-time was 5 to 10 minutes for each experiment.

5.3 Combining Tree Sub-Units: Whole Constituents and Parentheses

The first step of our program was to decompose the trees into sub-units which could then be recombined into new trees. As we have seen in chapter 4, it is possible to decompose trees into constituents, and to decompose constituents into parentheses. We performed experiments with each decomposition. The former has already been done in previous work, and the latter is a new experiment. We will now describe in details these two decompositions.

5.3.1 Whole Constituents

Whole constituents are the constituents that appear in the observed trees. Previous work used decomposition into constituents successfully. We used it as well in some of our experiments. We wanted to verify that the positive results obtained by previous work (Henderson and Brill, 1999), (Sagae and Lavie, 2006) could be reproduced with our parsers. We will give the result of basic experiments with whole constituents in section 5.7.1, and the results of more complex experiments in chapter 6.

5.3.2 Recombined Constituents

As we have seen in chapter 4, the idea for decomposing the whole constituents into parentheses and then recombining these parentheses into recombined constituents was to work with entities smaller than constituents. We wanted to test the hypothesis that using smaller entities would improve the ensemble learning. To make sure that this method could improve over methods using whole constituents, we calculated an upper bound score for experiments with decomposition of constituents into parentheses. This upper bound was superior to the upper bound of experiments with whole constituents. We will describe these calculations in details in section 5.4. We will now formalise the notion of recombined constituents.

We decomposed the whole constituents into parentheses. Formally, an opening parenthesis is a pair of (label, opening position), and a closing parenthesis is a pair of (label, closing position). We recombined pairs of opening and closing parentheses into recombined constituents if and only if the two following conditions were met:

- The label of the opening parenthesis is the same as the label of the closing parenthesis.
- The opening parenthesis occurs before the closing parenthesis: opening position <= closing position. (If the positions are equal, we have a one-word-long constituent).

There are two different types of recombined constituents: the *original* constituents, and the *new* constituents. The original constituents did already occur in an observed tree. Our rules do not specify that we cannot recreate those constituents. The only difference between the whole constituents and the original constituents is their origin. The whole constituents are directly extracted from the observed trees, whereas the original constituents are recreated by recombining their parentheses. The new constituents are

the recombined constituents that do not appear in any observed trees.

We weighted the recombined constituents and extracted the heaviest tree out of them. In section 5.6 we describe the weights, and, in section 5.5, we describe the algorithm that we used to build the trees.

5.4 Upper Bound

To make sure that the combination of parses from our parsers could be successful, we measured an upper bound, using the real output of our parsers. We used the parses of section 24 of the Penn Treebank. For every constituent, we know, for each parser, if the parser included or excluded the constituent from its parse tree. To calculate the upper bound, we used an oracle that predicts, for any constituent, which parser is right in including or excluding it. If every parser is wrong, then we do not have the answer, and there will be a mistake. The general idea is to count how many correct entities are in the output of at least one parser. These entities could also be in the final parse, if the oracle chooses them.

We calculated two upper bounds: one for a method using whole constituents and one for a method using recombined constituents. We could then compare the two methods to know if decomposing the constituents could improve the performance. Section 5.4.1 describes the upper bound calculations for whole constituents, and section 5.4.2 describes the calculations for recombined constituents.

5.4.1 Whole Constituents Upper Bound

We define constituents as *potentially correct* if they appear in the gold standard tree and in at least one corresponding observed tree. With the number of potentially correct constituents in a section, we can calculate the upper bound recall. If we could put all the potentially correct constituents in the final tree, then the recall would be:

$$\text{recall} = \frac{|\{\text{potentially correct constituents}\}|}{|\{\text{constituents in the gold standard}\}|} \tag{5.1}$$

With the observed trees of section 24, the upper bound recall on whole constituents is 93.9%.

To calculate an upper bound precision, we need a different denominator. The denominator in the precision calculations must be the total number of constituents in the result parse tree. This number can be further decomposed into two factors: the number of correct constituents in the parse tree, and the number of false positives. We will use the number of potentially correct constituents as the number of correct constituents, and we will follow the same intuition for the false positives. Since we calculate the best possible result, we will count as false positives only the constituents that must end in the final tree, and that we will not be able to discard. These constituents are the ones that are in every parser. The oracle will not be able to choose a correct parser for them. We don't count other false positives, because we assume that our oracle discards them. We call the false positives that must end in the final tree *unavoidable false positives*. We have:

$$\text{precision} = \frac{|\{\text{potentially correct constituents}\}|}{|\{\text{potentially correct constituents}\} \cup \{\text{unavoidable false positives}\}|} \tag{5.2}$$

The upper bound precision on whole constituents is 96.3%.

Both upper bound precision and upper bound recall are much higher than the result of the best parser (87.7% recall, 88.9% precison). These results are encouraging. They mean that a high improvement might be achieved by combining the parsers: the correct information exists. However, the problem of discriminating it from the noise remains.

5.4.2 Recombined Constituents Upper Bound

We also calculated an upper bound for recombined constituents. As with whole constituents, we use the notion of potentially correct constituents. These constituents are the ones that appear in the gold standard tree, and whose parentheses appear in at least one observed tree, not necessarily the same tree for both parentheses. We use the same definition of upper bound recall as for the whole constituents. This gives us an upper bound recall of 95.3%. This result is higher than the upper bound recall for whole constituents which supports the idea that recombining the constituents could lead to higher performances.

We did not adapt the concept of unavoidable false positives to the recombined constituents. The resulting precision would have been too low, because all recombinations of all whole constituents that are unavoidable false positive would also have been unavoidable false positives. Those are

clearly not inevitable, at least because they could have crossing parentheses one with another, and would then not form a tree. We did not calculate an upper bound precision for recombined constituents. We could expect the precision to decrease, because the technique can add a lot of noise. Despite this intuition, we decided to experiment with the technique for two reasons: we could not prove that the precision would effectively decrease, and even if it did, it was possible that a much higher recall compensated for it, still increasing the F-score.

5.5 The Tree Building Algorithm

We will now describe how we formed trees out of our weighted constituents. As we have seen in chapter 4, a set of constituents does not necessarily form a tree, and we wanted to output trees. So, when the constituents are weighted, whether they are whole or recombined constituents, we need to construct the heaviest possible tree out of them.

The weighted Cocke-Younger-Kasami (CYK) algorithm can output the heaviest tree out of a weighted grammar. This algorithm produces a solution in polynomial time: $O(n^3)$, where n is the length of the sentence to be parsed. We used a modified version of this algorithm. In section 5.5.1, we will describe the original CYK. In section 5.5.2, we will present a way to modify it to suit our needs. Finally, in section 5.5.3, we will describe the real implementation, which is a little different from the theoretical view.

5.5.1 The Cocke-Younger-Kasami Algorithm for Weighted Grammars

The CYK algorithm uses a technique called *dynamic programming*, so before describing the algorithm, we will introduce this notion.

Problems can often be divided into sub-parts. The direct way to solve some problems involves solving several times the same sub-problems. Suppose, for example, that we need to find the shortest path between point A and G in figure 5.2. A simple way to solve this is to calculate the length of every possible path and to select the shortest path. But several paths include C, and it would be more efficient to calculate the length of the shortest path from A to C, and to use it to calculate both the A to G path that passes by E and the one that passes by F. In dynamic programming, any sub-part of a problem is only solved once, so the shortest path from a point to another is

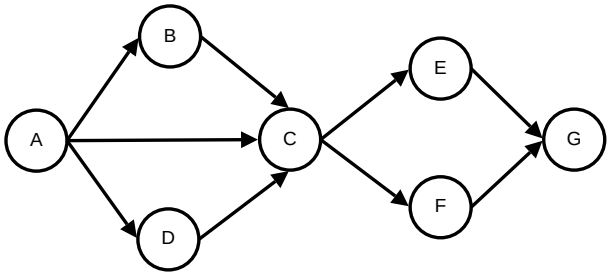


Figure 5.2: A path problem. It is possible to store the length of sub-paths, to prevent calculating them several times.

calculated once, then it is stored and reused for the other calculations. Suppose that the shortest path from A to all points but G is already calculated and stored. Here is how to calculate the shortest path from A to G:

$$P(G) = \min(P(E) + E \to G, P(F) + F \to G) \tag{5.3}$$

where $P(X)$ is the length of the shortest path from A to X, and $X \to Y$ is the length of the path form X to Y.

The dynamic programming approach applies to parsing. The problem here is, given a sentence and a binary weighted grammar, to build the heaviest tree that parses the sentence. The sub-problems are to find the best tree that parses each sub-part of the sentence. Consider the following sentence and weighted grammar.

1. Ada verified the code on the card.

S	$\longrightarrow$	NP VP,	w=3
NP	$\longrightarrow$	NP PP,	w=1
NP	$\longrightarrow$	Det N,	w=2
VP	$\longrightarrow$	VP PP,	w=1
VP	$\longrightarrow$	VP NP,	w=2
PP	$\longrightarrow$	Prep NP,	w=1
VP	$\longrightarrow$	V-verified	
NP	$\longrightarrow$	N-Ada	

CYK finds the heaviest tree that spans every sub-string of the sentence. It begins with the trees that span sub-strings of length 1, such as *Ada* or *code*, then it finds the other trees, the ones spanning a shorter sub-string before the ones spanning a larger sub-string of the sentence. It stops when it reaches the longest sub-string: the whole sentence. It uses a chart that stores the parses of every sub-strings. Figure 5.3 shows the chart for sentence

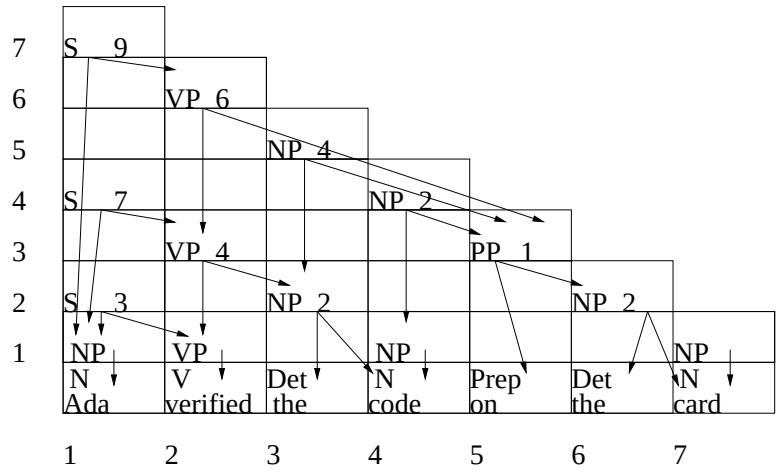


Figure 5.3: A CYK chart. Each cell contains the root node of the heaviest tree that spans it, the weight of this tree, and pointers to the two children of the root node.

1. The i^{th} cell (left to right) of the j^{th} line (bottom up) stores information about the sub-string of the sentence that starts at position i and is j words long.

Each cell stores the heaviest tree spanning the sub-string corresponding to it. It stores the root node of this tree, the total weight of the tree, and pointers to the two children of the root node. CYK fills the chart from left to right and from bottom up. First, it fills the lowest line which consists in the unary nodes on the terminals. To fill the higher cells, CYK considers all the possible binary partitions of the group of words that the cell spans. CYK compares each binary partition with the right handside of the rules. If the two elements of the partition are the two elements of the right handside of a rule then CYK makes the left handside of this rule the root of the tree stored in this cell. If several trees are possible with different root nodes, they are all added to the cell, but if they have the same root label, only the heaviest is stored. The weight of the tree is the sum of the weights of the two sub-trees and of the weight of the rule that produced the root node. There is no need to recalculate the two sub-trees: they are stored in the appropriate cells.

This algorithm produces the heaviest possible tree in time $O(n^3)$, where n is the number of words of the sentence. But it requires a binary grammar, and we have constituents which are not necessarily binary.

5.5.2 Modification of CYK for Constituents

We modified the CYK algorithm to fit our problem. We developed a version that directly uses constituents which do not have to be binary. Our version still finds the best solution in time $O(n^3)$.

Our intuition is to treat constituents directly, without using a grammar. The output of the algorithm must be the heaviest possible set of constituents with no crossing parentheses. We will only discard constituents if they *compete* with heavier constituents. Constituents compete with each other if they have crossing parentheses or if they have the same span.

To be sure that the set of constituents that the algorithm output is a tree, we need to meet two requirements: that there is no constituent with crossing parenthesis, and that there is at least one node that spans the entire sentence. We need to ensure that the first condition is met, but the second will always be met. The reason is that there always exists a constituent labelled *root* that spans the entire sentence. This constituent cannot have crossing parenthesis with another constituent. The only possible competitor for the root node would be a constituent that has the same span. And if this constituent wins, then we still have a tree, because the winning constituent will also span the entire sentence.

We will need to take special care of the *unary* constituents. A unary constituent has a single child. Formally two constituents C_1 and C_2 are *unary to each other* if C_1 and C_2 opening and ending positions are the same. Although the notion of unary to each other might seem odd, it is indispensable because in the constituent representation, if two constituents have the same span, it is impossible to know which one is the parent, and which is the child. Once the constituents are passed to the algorithm, unary ones are indistinguishable from competing constituents with the same span. They would compete with each other, and there would be no unaries in the resulting tree. Thus, we have to treat them before passing them to the algorithm. We merge unary constituents, retaining the information of which one is the parent and which one the child, and we pass the merged constituents to the algorithm, where they are treated as normal constituents. We will describe the exact method for merging unary in section 5.6.3.

The algorithm uses dynamic programming. We calculate the best set of constituents for each sub-string of the sentence. As in CYK, we calculate the best set of constituents for a group of words by taking the best union of the set of constituents that spans two sub-strings of our phrase, plus the constituent that best spans our phrase, if such a constituent exists. There

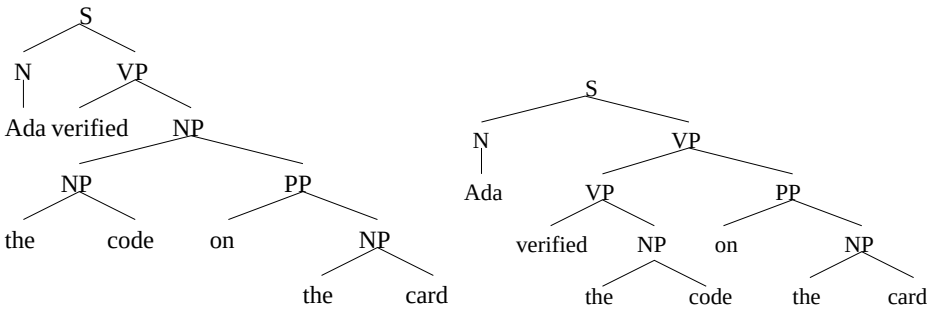


Figure 5.4: Two possible trees for a sentence. We will extract and weight the constituents before passing them to the tree building algorithm.

is no restriction on which constituent can be the parent of which other constituents, which leads each cell of the chart to be filled. The weight of a cell is computed by summing the weight of its set of constituents. The algorithm chooses randomly if two possibilities lead to the same weight.

For example, let us suppose that two parsers output the first tree in figure 5.4, and one parser outputs the second tree, for sentence 1. We extract the constituents from the trees and weight them; the weight of each constituents being the number of observed trees in which this constituent appears. We get the set of constituents in table 5.1.

Table 5.1: Set of constituents for the modified CYK

label	opening position	closing position	weight
S	1	7	3
VP	2	7	3
NP	3	4	3
PP	5	7	3
NP	3	7	2
VP	2	4	1
NP	6	7	3

Figure 5.5 shows the chart for combining these constituents into the heaviest possible tree. The first line is filled with the POS tag, then the other cells are filled with the constituent that exactly spans the sub-sentence corresponding to the cell, if it exists (in the upper left corner of the cell). The cell is also filled with references to the two cells that represent the two subparts of the original cell whose summed weight is the highest (in the lower left corner of the cell). If two decompositions produce the same weight, the algorithm chooses randomly. The weight of the cell is the sum of the weights

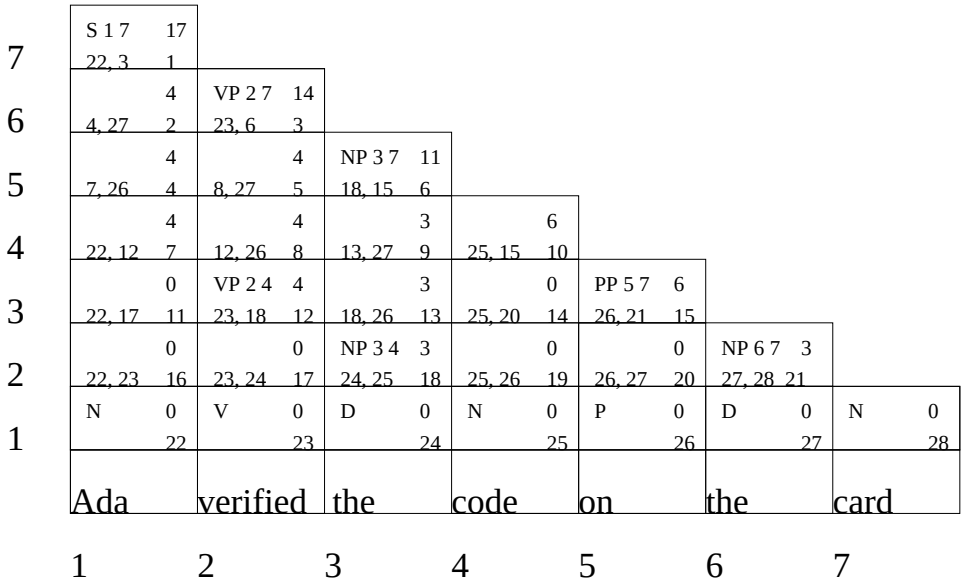


Figure 5.5: A chart for the modified CYK. Cells are filled with constituents and pointers to sub-partitions.

of its two sub-parts and of the weight of the constituent that spans the whole cell, if it exists. If such a constituent doesn't exist, then the weight of the cell is only the sum of its two sub-parts. Generally the weight of the i^{th} cell of the j^{th} line is calculated as follow:

$$W(i,j) = \max_{0<k\leq j}\{C(i,j) + W(i,k) + W(i+k, j-k)\} \tag{5.4}$$

Where $W(i,j)$ is the total weight of the tree that spans $(i, i+j-1)$, and $C(i,j)$ is the weight of the heaviest constituent that spans $(i, i+j)$ if it exists, and is 0 otherwise.

When the chart is full, the set of constituents that form the heaviest tree can be retrieved recursively:

$$\text{set}(C) = \text{const}(C) \cup \text{set}(C.\text{sub-part}_1) \cup \text{set}(C.\text{sub-part}_2) \tag{5.5}$$

Where C is a cell at position (i,j), const(C) returns the heaviest constituent that spans $(i, i+j-1)$ and $\emptyset$ if such a constituent does not exist. C.sub-part$_1$ and C.sub-part$_2$ are the two cells that constitute the best partition of C.

So the best set of constituents in our example can be found in table 5.2. The resulting tree has a total weight of 17.

Table 5.2: The set of constituent that the modified CYK outputs.

label	opening position	closing position	weight
S	1	7	3
VP	2	7	3
NP	3	4	3
PP	5	7	3
NP	3	7	2
NP	6	7	3

The complexity of the algorithm is $O(n^3)$. The algorithm needs to fill every cell of the chart: $\frac{1}{2}n^2$. For every cell of the chart it needs to test every $n-1$ possible partition. $\frac{1}{2}n^2 \cdot (n-1) \Rightarrow O(n^3)$.

We did not implement this algorithm. We implemented another version which is described in the next section.

5.5.3 Actual Implementation of the Tree Building Algorithm

As we have already stated, the two problems for applying CYK in our case is that we do not have a grammar, and that the constituents are not necessary binary. It is possible to use the CYK with a non constrained grammar, that is, allowing constituents to have any parent. It is also possible to binarise the constituents. The algorithm was mainly developed by Gabriele Musillo (personal communication).

We use a non constrained grammar: every constituent can be the parent of every two constituents that span a binary partition of the span of their parent. So, if the constituents are binary, we can use CYK.

To ensure that every constituent can be generated by two constituents, it is enough to binarise the observed trees before extracting their constituents. Trees are binarised by adding an auxiliary constituent with a special label every time the tree is not binary. Figure 5.6 shows an example of the binarisation process.

The label of the auxiliary constituent represents the labels of the sibling and parent node. All auxiliary labels are removed after the CYK has fin-

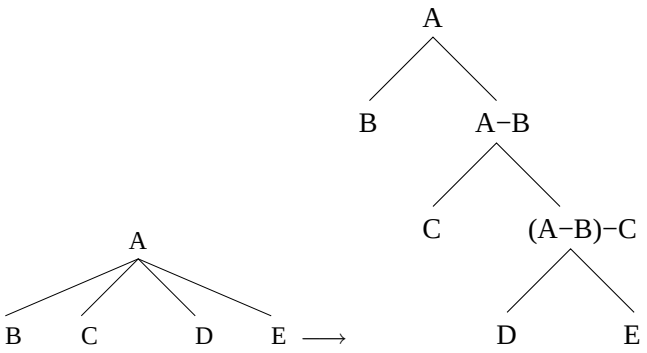


Figure 5.6: Binarisation process. The left tree is transformed into the right tree. The binarisation constituent can be removed to retrieve the original tree.

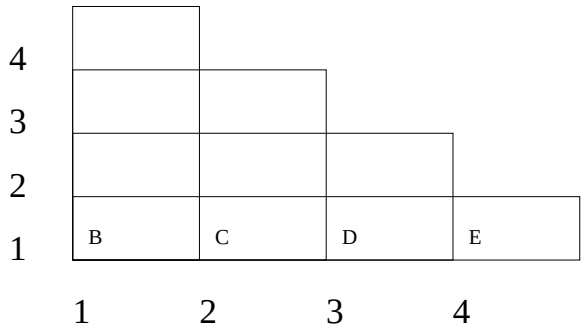


Figure 5.7: CYK chart for the non binary tree of figure 5.6. The algorithm fails to build a tree out of the constituents of the original tree for this sentence.

ished, to get the final tree. The binarisation constituents are only allowed to appear with their original parent and sibling.

The goal of the binarisation process is to ensure that every constituent has at most two children. Without this, the CYK algorithm as we use it would fail. For example, take the non binary tree in figure 5.6, assume that we use a non-constraining grammar, where each constituent can generate any combination of sub-constituents that covers the the whole span of the parent constituent. We would have the chart in figure 5.7.

So all cells of the second and third line are empty, because there are no constituent that spans exactly two or three words. If we take into account only binary partitions, we cannot fill the top cell, because no binary partition spans the whole span of the top cell. There are 3 possible binary partitions: cells 1,1 and 2,3, cells 1,2 and 3,2 and cells 1,3 and 4,1 (colon line). Each of these partitions contains at least one empty cell, which makes it impossible

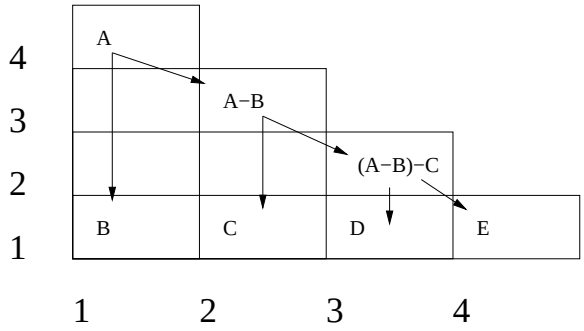


Figure 5.8: CYK chart for the non binary tree of figure 5.6. This time the algorithm can find the correct tree.

to fill the top cell. If we binarise the tree, however, we get the correct result. Figure 5.8 shows the CYK chart for the binarised tree of figure 5.6.

Note that the unary nodes must be merged before being passed to the CYK, as in the previous algorithm. The pseudo code for this experiment can be found in Appendix (Gabriele Musillo, personal communication).

5.6 Weighting Scheme

The purpose of the CYK algorithm is to build the heaviest tree out of a set of weighted constituents. The weight of a constituent should reflect the probability that this constituent is a true positive. We will now describe how we attributed weights to constituents.

(Sagae and Lavie, 2006) used a *vote weight*: the weight of a constituent is the number of parsers that output this constituent. Since they had good results, we also used this scheme to whole constituents. However we did not use it for recombined constituents: it is inapplicable because some of the recombined constituents do not appear in the output of any parser. We used two different schemes that are more adapted to recombined constituents: the *sum weight* and the *occurrence weight*.

5.6.1 The Sum Weight

The *sum weight* is similar to the vote weight, but it is better adapted to recombined constituents. We wanted the parsers to vote on opening and closing parentheses rather than on whole constituents. We made the hy-

pothesis that the number of occurrences of a parenthesis in the parses is proportional to the probability that this parenthesis is a true positive.

We summed the weight of the opening and closing parentheses to get the weight of the whole constituent. Here is the equation of the sum weight:

$$\begin{gathered} weight(\text{label}, \text{opPos}, \text{cloPos}) \\ = \sum_{i=1}^{3} (nbOp(p_i, \text{label}, \text{opPos}) + nbClos(p_i, \text{label}, \text{cloPos})) \end{gathered}$$

Where p_i are the parsers, and nbOp (respectively nbClos) returns the number of opening (closing) parentheses at position opPos (cloPos) in the output of the parser p_i.

An important difference between this measure and the vote weight is that here the parsers do not output a single vote for each parenthesis, they output the number of time each parenthesis appears in their parse. The difference relies on the fact that a parenthesis often occurs multiple times at a given position with a given label in an output tree, whereas the same is impossible for whole constituents.

Let us examine the behaviour of the sum weight function. The smallest weight is $w = 2$. That is $\sum_i \text{nbOp}(p_i, \text{lab}, \text{pos}) = 1$ and $\sum_i \text{nbClo}(p_i, \text{lab}, \text{pos}) = 1$. Both parentheses must occur in at least one parser output for the constituent to be created (but both parentheses need not appear in the same parser output). The maximum weight, however, is not theoretically bounded. It is bounded only by the actual data.

5.6.2 The Occurrence Weight

We experimented with the sum weight on whole constituents and got poorer results than with the vote weight (this experiment is more documented in section 5.7.2). This led us to design a measure closer to the vote weight for recombined constituents. A weakness of the sum weight is that it is unbounded. Constituents can get a very high weight, even if they appear in few parsers. For example, a recombined constituent that does not appear in any parse can compete and win over a constituent that appears in every observed tree. We believe that this behaviour could be a cause for bad results, and we tested this hypothesis by experimenting with another measure, which is bounded, and has a behaviour more similar to the vote measure: the occurrence weight.

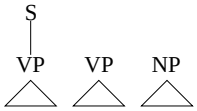


Figure 5.9: Unary constituents. The left tree includes a unary constituent which will compete with the constituents in the middle and in the right trees.

$$weight(\text{label}, \text{opPos}, \text{cloPos}) = \sum_{i=1}^{3}(occOp(p_i, \text{label}, \text{opPos}) + occClos(p_i, \text{label}, \text{cloPos}))$$

Where $\text{occOp}(p_i, \text{label}, \text{opPos})$ (resp. $\text{occClos}(p_i, \text{label}, \text{cloPos})$) returns 1 if an opening (resp. closing) parenthesis with this label appears at least once at position opPos (cloPos), in the parse of parser p_i, and returns 0 otherwise. This function takes into account only the fact that the parenthesis occurs in the parse tree, not the number of time it occurs.

The expected behaviour of this measure is closer to the behaviour of the vote measure. The minimum is occWeight = 2, when one parser proposes the opening parenthesis and another (not necessarily the same) proposes the closing parenthesis. The maximum is occWeight = 6, where all parsers propose both parentheses (but not necessarily as part of the same constituent). Consensual whole constituents always get the maximum weight.

Although the occurrence weight is closer to the vote weight than the sum weight, it is not equivalent. If we apply this measure to whole constituents, we do not get the same results as with the vote measure. The difference is that with the occurrence weight parsers that do not propose a whole constituent can still influence its weight by proposing only one of its parenthesis.

5.6.3 Unary Constituents

The unary constituents are merged before entering the tree building algorithm. This process creates *complex* constituents. We must use a special weight for the complex constituents. Here is an example of this merging process. Each tree in figure 5.9 represents a sub-tree returned by a different parser. The right tree includes a unary constituent S which must be merged with its child VP.

If we merge the unary constituents, we get three different constituents: S-VP, VP and NP. The first one is a complex constituent. We could give the same weight to all three constituents. They would then all have a weight of

2. But, intuitively, these constituents are not independent from one another. The VP should contribute to the weight of the S-VP, because, if the correct answer is VP, outputting S and VP would increase the score, compared to outputting NP. Reversely, if the correct answer is S and VP, outputting only VP is better than outputting NP. We weigh the constituents in the following way:

Let C be a complex constituent resulting from the merge of c_1, c_2, ..., $c_{\text{length}(C)}$. Let X_1, ..., X_n be complex constituents resulting from the merge of c_i with other constituents. Then

$$\text{weight}(C) = \text{vote}(C) + \frac{\text{vote}(c_1)}{\text{length}(C)} + ... + \frac{\text{vote}(c_n)}{\text{length}(C)}$$
$$\text{weight}(c_i) = \text{vote}(c_i) + \frac{\text{vote}(X_1)}{\text{length}(X_1)} + ... + \frac{\text{vote}(X_n)}{\text{length}(X_n)}$$

This can be seen as a grade system: each parser gives a grade to each constituent. A parser gives a whole grade to the constituents that it proposes, a partial grade to the constituent from which a part appears in its output, and a null grade to other constituents. The vote function can be one of the preceding measures: sum weight or occurrence weight.

5.7 Experiments and Results

In this section we describe our first experiments with whole and recombined constituents, with the sum weight or the occurrence weight.

5.7.1 Baseline: Whole Constituents

We did a baseline experiment, which is very similar to previous work. This experiment serves as a comparison for the other experiments. In this experiment we used only whole constituents.

We put of all the constituents that appear in at least one parser output in a bag. We weigh all the constituents before passing them to the tree building algorithm, which outputs the heaviest possible tree. If a constituent competes with another in the tree building algorithm and loses, it does not appear in the final tree.

We performed this experiment with the three different weighting schemes: the vote weight, the sum weight and the occurrence weight. Table 5.3 shows

Table 5.3: Results of the simple experiment with whole constituents.

	Precision	Recall	F-score
Fun. Parser (baseline)	88.9	87.7	88.3
Vote Weight	86.9	90.5	88.7
Sum Weight	86.2	89.6	87.9
Occurrence Weight	86.4	89.8	88.1

the results, together with the performances of the best of the three parsers: the function parser.

We get the best results with the vote weight, but this measure can't be used in further experiments, because it doesn't apply to recombined constituents. The decrease in performance with the other two measures shows that taking partial constituents into account adds noise. It seems that the fact that parts of a constituent appear in other parsers does not increase the probability that this constituent is a true positive.

The occurrence weight gives slightly better results than the sum weight. This does not contradict our intuition that the occurrence weight is better than the sum weight and closer to the vote weight, but the difference is not as high as we might have expected. Further experiments will confirm this slight superiority.

5.7.2 Recombined Parentheses

We will now describe our simplest experiment with recombined constituents. It is similar to the baseline, but this time we used a set of recombined constituents instead of whole ones.

We experimented with the sum weight and the occurrence weight. We calculated the number of new constituents and of correct new constituents in the results. The new constituents are constituents that appear in the final tree but not in the output of any original parser. The correct new constituents cannot be found with a method that only uses whole constituents. Note that the number of new constituents is not the number of constituents that were added to the output of an original parser. The differences between an original parser output and the output of the experiment can be the following: deleted constituents, that is constituents that are in the original parser but not in the experiment; added original constituents, that is con-

Table 5.4: Results of the simple experiment with recombined constituent.

	Prec.	Rec.	F-score	New Const.	Corr. New
Fun. Parser (baseline)	88.9	87.7	88.3		
Sum Weight	80.4	87.1	83.6	2218	89
Occ. Weight	80.3	87.3	83.6	2425	95

stituents that are in the output of an original parser but not in the output of the original parser considered and that are also in the experiment output; and new constituents, that is constituents that are not in any original parser output, but that are in the experiment output. Table 5.4 shows the results.

These results are worse than the baseline. New constituents add many more false than true positives. Neither the sum nor the occurrence weight is able to discard effectively the false positives. For the occurrence weight, we believe that constituents recombined out of two consensual constituents are a large proportion of the false positives. These constituents always get the maximum weight, although, intuitively, they are not necessarily very good, and they can win against constituents that appear in the output of two parsers, for example.

5.8 Conclusions

The recombination gives poorer results than the baseline and the original parsers. Our intuition that the recall would be improved is not confirmed.

The decrease in precision means that the recombination adds false positives to the results. This is not surprising, since the process creates many more candidates than the baseline and can give high weights to constituents that appear in few parsers.

The decrease in recall means that the false positives not only end up in the final parse, but also win against true positives, discarding them from the final tree. This result suggests that the weighting scheme is inadequate to represent the probability of the constituents. All the original parsers produce a high precision output, and the large majority of the constituents that they propose are true positives. This is not true for the recombined constituents. For those constituents, we have no guarantee that they are true positives.

We must choose recombined constituents more restrictively. We developed several experiments to restrict the combination of constituents, and the access of false positives to the final tree. We will describe these experiments in the next chapter.

Chapter 6

Developing a More Complex System

As we have seen in the previous chapter, the results of our first experiments did not meet our expectations. In fact, the experiments with recombined constituents achieved poorer results than the individual parsers. We saw that using recombined constituents adds much noise, and that a very large majority of the new constituents in the results were false positives. It became necessary to find a better way to select the recombined constituents. In this chapter we will describe more methods to increase the performance, and to discriminate true positives form false positives.

In sections 6.1, 6.2 and 6.3, we will describe three methods to produce fewer recombined constituents. We will also examine two methods to restrict the access of false positives to final trees in sections 6.4 and 6.5. Finally, we will describe a method to discriminate true positives from false positives with a support vector machine classifier, in section 6.7.

6.1 Recombining only Constituents that Share a Non-Empty Intersection

As we have seen, our early experiments with recombined constituents showed that adding new constituents tended to degrade the performances. To solve this problem, we tried to limit the number of new constituents in the resulting trees, in two different fashion. We limited the creation of new constituents, and we limited their accessing the final parse. We will now describe the first of three methods to limit the creation of new constituents.

Table 6.1: Results of recombining only constituents that share a non-empty intersection

	Prec.	Rec.	F-score	New Const.	Corr. New
Fun. Parser	88.9	87.7	88.3		
Simple	80.3	87.3	83.6	2425	95
Sum Weight	84.6	88.8	86.7	640	37
Occ. Weight	84.8	89.2	86.9	648	34

It happens that two parsers propose a constituent with a certain label at a position slightly different from a parser to the other. In this case we can assume that the two parsers "mean" the same constituent. There must be a constituent of this label close to the position where both parsers propose it, but its exact span is uncertain. The fact that several parsers propose slightly different spans is a clue that the exact span is uncertain, and the right constituent might be different from the proposal of each parser. But even if both parsers are wrong, it is possible that they are both close to the solution. The fact that the parsers output constituents that are close but not equivalent is a clue that there is uncertainty on the constituent that should be here. We wanted to add noise where there is uncertainty because this would generate more possible constituents among which the correct one would hopefully be. We can achieve this by recombining the concerned constituents.

We also wanted to avoid recombining constituents that are linearly far away from each other in the tree. Recombining these constituents would make new constituents that are much bigger than the original ones and that make little sense.

Formally, we decided to recombine two constituents only if they share a non-empty intersection, that is, if they are close to each other. This situation can happen in two cases. Either the constituents have crossing parentheses, or one constituent is embedded in the other.

This experiment is very similar to the simple recombined constituents experiments described in section 5.7.2, except that this time we only recombined constituents that share a non-empty intersection. The other constituents are kept whole. We use the same weighting scheme for the whole constituents and for the recombined ones.

Table 6.1 shows the results. We also included the results of the simple experiment with the occurrence weight with recombined constituents (5.7.2), which constitutes a baseline for recombined constituents. We also included

the results of the best individual parser, for comparison.

Altough it remains lower than the baseline with whole constituents, the F-score increases compared to the simple experiment with recombined constituents. This result shows that we add less noise than in the recombined constituents baseline. Creating fewer new constituents lead to fewer false positive new constituents in the results and to an increase in performance. However, the new constituents still add more false positives than correct constituents, and the percentage of correct new constituent is still far less than 50%.

6.2 Recombining only Constituents Voted by Few Parsers

In our experiments with whole constituents we made the hypothesis that the more parsers proposes a constituent the more probable it is that the constituent is a true positive. This kind of hypothesis is at the core of ensemble learning, and it gave good results in our experiments. Here we use something similar to discriminate constituents that should be recombined from constituents that should be used directly as whole.

Consensual constituents are most probably true positives and we don't want to add noise to them, so we will use them directly. We want, however, to add some noise to constituents that are uncertain. This can be done by recombining them. This creates more possibilities and, hopefully, more correct constituents. We will only recombine constituents that few parsers propose, since they are the less certain.

We did two experiments with recombining only constituents proposed by few parsers. In the first experiment, we recombined only constituents that were proposed by one parser. Since we have three parsers, it is equivalent to recombining only constituents that got a minority of votes, and that are most uncertain.

In the second experiment we kept as whole only the consensual constituents that were voted by all parsers, and are probably true positives. We recombined the other constituents that were proposed by one or two parsers.

For comparison, we also tried to recombine only consensual constituents and only constituents that are voted by exactly two parsers. We expected the performance to drop on those experiments. We used only the occurrence

Table 6.2: Results for recombining constituents voted by few parsers

const. recombined	Prec.	Rec.	F-score	New Const.	Corr. New
Fun. Parser	88.9	87.7	88.3		
simple	80.3	87.3	83.6	2425	95
vote=1	86.0	89.7	87.8	153	5
vote=2	86.3	89.8	88.0	68	9
vote<3	85.8	89.7	87.7	299	20
vote=3	81.7	87.8	84.6	1706	63

weight, since it gave better results in the previous experiments. Table 6.2 shows the results together with the baselines.

As we expected, the performance dropped when we recombined the consensual constituents. We believe that the reason is that consensual constituents tend to be already true positives and that recombining them adds noise, which results in a decrease in precision. The recall also decreases, because more new false positives can access to the final parse, "taking the place" of true positives.

The performances are about the same for constituents recombined out of constituents voted by one parser or by two parsers. We believe that the slightly better performance of the experiment with recombining only constituents voted by two parser is the result of the fact that there are a fewer constituents voted by exactly two parsers than voted by exactly one (3362 vote=2, 4937 vote=1).

The number of new constituents is small in both experiments, probably because the new constituents get a small weight, since their weight is the mean of the weights of their parents, which is proportional to the number of parsers that proposed them.

The results were better than on the previous experiment were we recombined only constituents that shared a non-empty intersection. Note that in the previous experiments it was possible to recombine consensual constituents if one was embedded in the other. Those recombinations might have generated false positives, since they produced constituents that got a big weight, and that might not be intuitively very good.

We tried to combine the two experiments. We recombined constituents that shared a non-empty intersection and that were voted by few parsers.

Table 6.3: Results for recombining constituents voted by few parsers and that share a non-empty intersection.

const. recombined	Prec.	Rec.	F-score	New Const.	Corr. New
Fun. Parser	88.9	87.7	88.3		
simple	80.3	87.3	83.6	2425	95
vote=1	86.1	89.7	87.9	108	4
vote=2	86.4	89.9	88.1	18	3
vote<3	86.0	89.7	87.8	187	14

Table 6.3 shows the results, for these experiments together with the baselines.

This experiment led to a slight over the previous one. Very few new constituents ended in the final parses. Of those, only a small percentage was correct. It seems that the number of new constituent is inversely proportional to the performances. We believe that the reason is that new constituents add more noise than true positives, and that, even when restricting the number of constituents that we recombine, we were not able to correctly discriminate the correct new constituents from the false positives.

6.3 Recombining only Constituents that Share a Common Spine

We performed another experiment similar to the experiment described in 6.1 where we recombined constituents that shared a non-empty intersection. We still wanted to recombine two constituents if they were similar. We didn't want to recombine constituents that were too dissimilar, because, intuitively, the resulting constituents would not make a lot of sense. In the first experiment we recombined only constituents that are close to each other in their position on the sentence. In terms of tree, this meant that they were linearly close. For example the constituents C2 an C4 in the two trees of figure 6.1 are linearly close. They appear near to each other in the sentence and have crossing parentheses.

Altough they have crossing parentheses and are linearly close, C2 and C4 from figure 6.1 are structurally far away from each other. Specifically, they have different ancestors. In this experiment, we wanted to recombine constituents that are not only linearly but also structurally close. To represent this similarity between constituents we use the concept of *spine*. The spine of a constituent is the path from a node to the root of this constituent. In

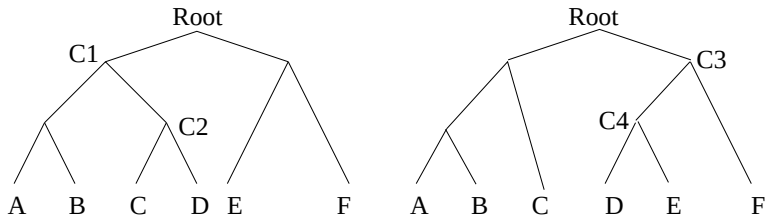


Figure 6.1: Two constituents that are linearly close, but structurally far away. C2 and C4 have crossing parentheses but they have different ancestors.

Table 6.4: Results of recombining only constituents that share a common spine.

	Prec.	Rec.	F-score	New Const.	Corr. New
Fun. Parser	88.9	87.7	88.3		
simple	80.3	87.3	83.6	2425	95
Sum Weight	85.2	89.3	87.2	402	28
Occ. Weight	85.3	89.5	87.3	445	29

the previous tree for example the spine of C2 is Root-C1 and the spine of C4 is Root-C3. We did this experiment by recombining only constituents that shared a entire common spine. Those constituents are linearly and structurally close to each other. Table 6.4 shows the results together with the baselines.

As in the previous experiment, the proportion of correct constituents among the new constituents is very low. We were not able to discriminate true from false positives in new constituents by restricting the possible parents. In the following experiments we will try to increase the proportion of correct new constituents not by choosing fewer possible parents, but by better choosing, among the recombined constituents, those that will be part of the final parse.

6.4 A Threshold on Weight of Constituents that Enter the Tree Building Algorithm

As we have seen in the previous experiments, we have many false positive constituents in the final parse, especially among new constituents. Some of these false positives have a small weight. Constituents with a small weight can be part of the final parse in two circumstances. First, it can happen that several constituents with a small weight compete and win against fewer

constituents with a bigger weight. For example, a constituent with weight 4 can lose against two constituent with weight 2 that compete with it. Second, some constituents do not have any competitor. They end up in the final parse independently of their weight which can be very low.

To prevent this phenomenon where constituents with a low weight enter the final parse tree, we tried to apply a filter on weight before the tree building algorithm. Constituents whose weight is lower than a threshold are discarded, and only the others enter the algorithm.

We build a set of all constituents from any of the parsers. We paired each constituent with the other constituents bearing the same label, and we created the recombined constituents. Then we calculated the occurrence weight of each constituent. We applied a threshold to the constituents discarding those which weight is inferior to 4. This also have the effect of discarding original constituents which vote weight is inferior to two: the minority original constituents. We then passed the remaining constituents to the tree building algorithm.

Simply discarding the constituents that did not pass the threshold, would create problems for running the tree building algorithm. Some of the discarded constituents could have been necessary to have a binary grammar. Without those constituents, it can be impossible for the algorithm to construct a tree. Therefore, we replaced all original discarded constituents with *ghost constituents*. These constituents have the same span as the discarded ones they replace, but they have a special label, and a null weight. They are trimmed out of the final tree returned by the tree building algorithm.

We did several experiments with thresholds. The first one was with whole constituents only. We wanted to see if we got better results than our baseline with whole constituents. We used the vote weight, and discarded the minority constituents. In the second experiment we recombined all the constituents. And in the third experiments we combined the threshold restriction with a restriction on which constituents could be recombined: we recombined together only constituents that shared a non-empty intersection like in 6.1.

Table 6.5 shows the results for these experiments, together with the baselines and the results of the simple experiment with vote weight and whole constituents (5.7.1).

The first experiment, which involved only whole constituents gave an F-

Table 6.5: Results for experiments with a threshold.

	Prec.	Rec.	F-score	New Const.	Corr. New
Functional Parser	88.9	87.7	88.3		
Simple Whole	86.9	90.5	88.7		
Simple Rec.	80.3	87.3	83.6	2425	95
Whole Consts.	91.4	88.5	89.9	0	0
All Rec.	82.0	86.7	84.3	2379	93
Non-empty Inter. Consts.	86.8	88.6	87.7	617	30

score more than 1 point better than the baseline. The precision is improved and the recall decreases. This is not surprising, since the goal of this experiment was to remove uncertain constituents. The improvement shows that we removed more false positives than true positives. Note this experiment is very similar to the work of (Henderson and Brill, 1999) and (Sagae and Lavie, 2006).

The experiment with recombined constituents led to an amelioration over the recombined constituents baseline. But the performances are lower than in the previous experiments and much lower than the performances of the individual parsers. Once again, the new constituents add much more noise than true positives.

6.5 Giving Infinite Weight to Some Constituents

In the previous experiment we got an improvement over the baselines by letting only the majority whole constituents enter the tree building algorithm. When we let other constituents, the performance decreased. So we wanted to leave no competition for the majority constituents, and to let the new constituents compete only among themselves and against constituents voted only by one parser. We didn't want the new constituents to be able to win against majority constituents because we saw that the new constituents tended to add noise, whereas the majority whole constituents alone led to good results. We implemented this intuition by giving an infinite weight to majority whole constituents. This meant that all of them would end up in the final parse.

We used the occurrence weight, and did two experiments: one with recombining all constituents, and one with recombining only constituents that shared a non-empty intersection. Table 6.6 shows the results together with

Table 6.6: Results for giving infinite weight to some constituents.

	Prec.	Rec.	F-score	New Const.	Corr. New
Fun. Parser	88.9	87.7	88.3		
simple	80.3	87.3	83.6	2425	95
All Rec.	83.3	90.4	86.7	1434	50
Non-empty Inter. Consts.	86.0	90.4	88.2	405	24

the baselines.

Since all majority constituents end up in the final parse, the recall must be at least as high as that of the experiment with whole constituents and a threshold (6.4). Actually, the recall is much higher. This means that there were many true positives among the constituents voted by only one parser and the new constituents. However, the recall is slightly lower than the recall of the baseline with whole constituents (5.7.1). This means that some false positives among the new constituents took the place of correct minority whole constituents.

The precision is still not satisfactory, and the new constituents still add more noise than true positives. This results in an F-score lower than the F-score of the individual parsers, and much lower than the experiments with whole constituents only.

6.6 Combining Threshold and Infinite Weight

Both previous experiments led to an increase in performance compared to the recombined constituents baseline. The experiment with threshold and whole constituents (6.4) also led to improvement over the baseline for whole constituents, and to a score much better than the score of the best individual parser. We combined the two previous experiments into a new experiment.

We recombined every constituents with every other constituents that had the same label. Then we treated the recombined constituents in different fashions. The majority original constituents are the constituents that appeared in at least one parser output and that have a vote weight bigger than one. Those constituents were given an infinite weight as in the infinite weight experiment (6.5). The minority original constituents and the new constituents were weighed with the occurrence weight. Then we applied a threshold on those constituents. We tried two different threshold. In the

Table 6.7: Results for combining threshold and infinite weight

Threshold	Prec.	Rec.	F-score	New Const.	Corr. New
Fun. Parser	88.9	87.7	88.3		
simple	80.3	87.3	83.6	2425	95
w < 4	85.0	89.8	87.3	1388	48
w < 6	87.3	89.0	88.1	1084	40

first experiments we discarded all minority original constituents and new constituents that got an occurrence weight strictly inferior to 4. In the second experiment the threshold was 6. We wanted to use as a complement to majority original constituents the minority constituents and the new constituents that had a weight above the threshold and we hoped to increase the recall with them. Table 6.7 shows the results together with the baselines.

The recall must be as high as the one of the experiment with a threshold and whole constituents, since all majority original constituents end up in the final parse. It is also bigger than the recall of experiments with recombined constituents and a threshold (6.4). This suggest that in the threshold experiment with recombined constituents, some false positives took the place of correct majority original constituents, and that this effect was cancelled by the infinite weights.

The recall is also lower than the recall of the experiment with infinite weights 6.5, because some correct constituents were discarded by the use of a threshold. Not surprisingly, this effect is more present with a bigger threshold.

The precision is higher than the precision of the experiment with all constituents recombined and infinite weights, because the threshold discards some false positives. By combining both experiments, we achieved a recall better than the recall of the threshold experiment, and a precision better than that of the infinite weight experiment. The F-score, however, is still not satisfying, and the new constituents still add more noise than correct constituents.

6.7 Using a Support Vector Machine for Classification

We did another experiment to see if it was possible to use other features to discriminate true positives from false positives among the whole constituents. We wanted to see if other parameters, or combinations of parameters would give better clues.

As we have seen in chapter 2, support vector machines are used to classify entities based on a set of clues. If a clue is not discriminant, an SVM can ignore it. This makes SVM a natural choice to explore how parameters could help to classify constituents into two classes: true positives and false positives.

To train our SVM, we used libSVM (Chang and Lin, 2001). LibSVM provides a script (easy.py) that automatically scales data, and selects the best parameters for the RBF kernel. This script has been been shown to be appropriate for most cases ((Hsu et al., 2003)).

We wanted to use as many parameters as possible, because useless ones could be ignored by the algorithm. However, we had to take a not too long list of parameters, because, as each parameter makes the search space bigger, the computation time of the SVM training can get very big.

We used the following list of feature: the length of the sentence in words, the length of the constituent in words, the label of the constituent, the vote of each parser. We included the vote of the parsers because it was the only feature used for our previous experiment and that it gave good results for whole constituents. Our intuition for including the label was that some parsers might be better at some labels than other. Similarly we included the length of the sentence and the constituents because we tough that some parsers might be better at parsing longer sentences or longer constituents.

With this set of feature, the SVM could recreate the voting scheme since the parsers votes are a parameter. The algorithm could ignore all other parameters, sum the votes and discard constituents that got only one vote. This would be similar to the method for whole constituents with threshold described in 6.4.

As a baseline we trained the classifier with only the parsers vote as features. We believe that in such a case, the classifier would learn to classify constituents that get a majority of vote in the true positive class and the

others in the false positive class. We don't believe it would take into account which parser issued the vote, because the parsers have very close performances, and the performance of combining them is higher than the performance of any individual parser, and giving priority to one of them would effectively select only its output which is poorer than the combination.

As training data we chose the first section of the Penn Tree Bank. We took only 9900 constituents out of this section for training. We took a constituent every 10 constituents of the section. On every 100th constituent we did not put it in the training data but in the testing data. This made a testing corpus of 1100 constituents. We used the same testing and training corpus for all experiments.

For the baseline we got an accuracy of 91.8. That is, 1008 constituents were classified successfully out of 1100. For the real experiment with all the features, we got exactly the same result: 1008 constituents out of 1100 were classified successfully. This strongly suggests that the algorithm learnt to ignore all features but the parsers vote.

We believe that this experiment showed that none of the feature that we used is useful to discriminate the true and false positive constituents. We were not able to find a better way to discriminate them, despite using many features.

6.8 General Discussion

None of the experiments with recombined constituents did achieve better performances than the individual parsers. The best F-score was 88.2% for the experiment with infinite weight in which we recombined only constituents that shared a non-empty intersection (6.5). This experiment also achieved the best recall (90.4%), because all majority constituents, which tend to be true positives, ended up in the final parse. Minority constituents and new constituents could not take the place of majority constituents, but could still access to the final parse, resulting in a better recall. The best precision (87.3%), however, was achieved by the experiment with infinite weight and a high threshold (6.6), because the threshold discarded many false positives.

There are about 25'600 constituents in the gold. This means that adding 256 correct constituents would increase the recall by 1%. It was our goal to increase the recall by adding new constituents, but in each experiment the

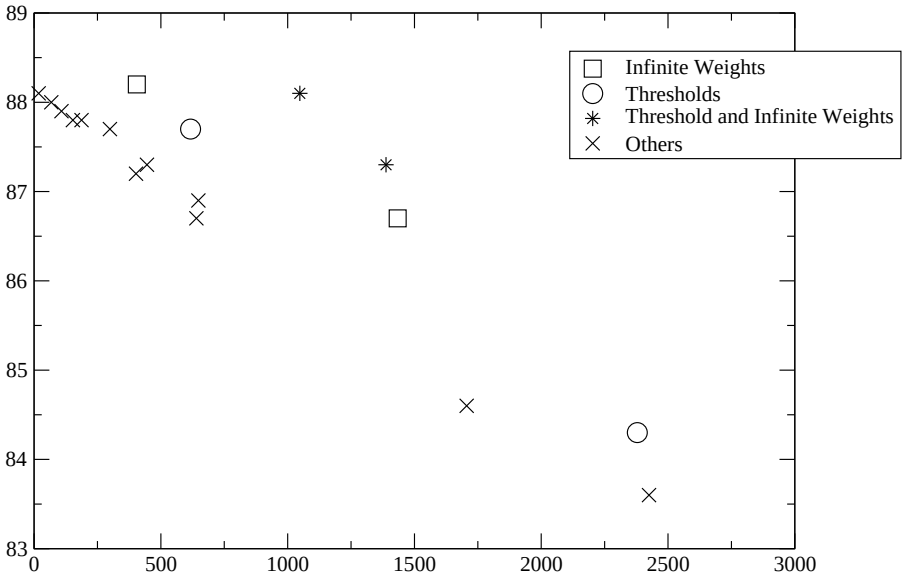


Figure 6.2: Our experiments with their F-Score and the number of new constituents in the final parses of section 24.

number of correct new constituent was far less than 256.

A constant of all previous experiments is that it seems that the more new constituents ended up in the final tree, the worst the performance was. We plotted the number of new constituents and the F-score to have a better idea of this phenomenon. Figure 6.2 shows this plot, where each point represents an experiment with recombined constituents.

The experiments with thresholds and infinite weights can achieve a better F-score than the others, given the number of new constituents. They chose better the constituents that ended up in the final parses. But even in those experiments the number of correct new constituents was much lower than the number of false positive new constituents.

To ensure that the bad results were not due to a bias of the evaluation system, we also calculated the performances on entire trees. These perfor-

mances were also much lower than the individual parsers: the functional parser achieves 27.7% entirely correct trees, the best of our experiments that involves recombining constituents is 21.3% correct trees. This result is achieved by several experiments such as when we recombined only constituents that were voted by exactly one parser. The worst experiment on this level was the simple experiment with the occurence weight: 9.5% of entirely correct trees. The experiment with whole constituents and a threshold was the only one that surpassed the baseline with 28.3% entirely correct trees. Recombining the constituents did not lead to any improve in performance, despite the number of experiments. All the percentages for all experiments can be found in appendix.

When recombining constituents we destroyed some information that the parsers output. We lost the information of which parenthesis should form a constituent with which other parenthesis. We were not able to recreate this part of the work of a parser, which is based on a complex algorithm, with the simple algorithm that we used.

If the weight that we used are good estimators of the probability of occurrence of a single parenthesis, when we summed the weights of two parentheses to get the weight of the recombined constituent, we lost the important information of the probability that both parentheses belong to the same constituent. Here is a possible configuration of parentheses:

(	A	B	)	(	C	D	)
1			2	3			4

In this configuration and with our weighting scheme, if all parentheses appear in the same number of parsers, the weight of the constituent formed of parenthesis 1 and 2 and the weight of the constituent formed of parentheses 1 and 4 must be the same, even if all parsers proposed the former constituent and none proposed the later. The recombination of parentheses destroyed information that we could not reproduce with a simple algorithm.

6.9 Conclusions

In this chapter we described a number of experiments that we tried to discriminate correct and false positive constituents, especially among the new constituents. We could not find a way to discriminate them. No experiment got more correct new constituents than false positive new constituents. In every experiment there was far more false positives than true positives among the new constituents. The only experiment that got higher performances

than the whole constituents baseline, the experiment with whole constituents and a threshold described in 6.4, did not involve new constituents and was very similar to previous work. Actually, no experiment involving new constituents did get a result better than the result of the function parser. The new constituents always added more noise than true positives.

Chapter 7

Conclusions

In this work we recombined the output of three statistical parsers to improve their performance. We decomposed the output of the parsers with several levels of granularity: into constituents and into parentheses. We performed several experiments with several level of decomposition of the parsers output (constituents and parentheses), several ways of restricting the creation of new false positives and their access to the final parse and several weighting schemes.

We found that our parsers were diverse enough to be combined, despite the fact that they are all based on the same algorithm, and that their only difference is the granularity of the information that they were trained on and that they can output. Our upper bound suggested that the parsers could be successfully combined, and the results of the experiments with whole constituents (5.7.1, 6.4) showed that the performance could indeed improve.

The experiment with the best F-score was the experiment with whole constituents and a threshold (6.4). This experiment was very similar to previous work. We saw that discarding minority constituents resulted in a better precision and a poorer recall. The drop in recall was however compensated by a much better precision, resulting in a better F-score.

Since we had several constituents with crossing parentheses in the set of constituents we wanted to recombine, we had to use an algorithm to ensure that the constituents in the final parse would form a tree. We also wanted this tree to be the heaviest possible tree given the weight of the constituents. We adapted an existing algorithm to out needs: the weighted Cocke-Younger-Kasami algorithm.

Despite several attempts we were unable to obtain good results with con-

stituents recombined out of parentheses. We failed in finding a weight that could be applied to recombined constituents and that would be proportional to the probability that the constituent was a true positive. We used two weighting schemes for recombined constituents (the sum weight in section 5.6.1 and the occurrence weight in section 5.6.2). These weights degraded the performances when used on whole constituents, but the voting scheme that gave good results could not be directly transposed to new constituents.

Our hypothesis that recombining constituents would improve the recall was not correct. False positives ended up in the final trees and took the place of true positives, degrading not only the precision, but also the recall.

Even if we could find the correct parentheses, based on the votes of the parsers, we could not find the correct constituents to which they belonged. So that even if a majority of correct parentheses ended up in the trees, the trees were not necessarily entirely correct nor contained correct constituents. The ensemble learning was probably successfull at the parentheses level, that is the set of parentheses of the final tree might be mostly correct, but we added some information to the raw parentheses by putting them together into constituents. The information concerning the matching of parentheses is important to the evaluation, but it was added without taking probabilities into account.

If the we wanted the recombination to work better, we would need to integrate probabilities that two parentheses belong to the same constituents. But it is possible that finding these probabilities would be a task of complexity similar to the entire parsing process itself.

Appendix A

Pseudo Code for the Modified CYK

This pseudo code was mostly written by Gabriele Musillo.

Let the constituents (recombined or whole) be tuples of the form (X, i, j, w) where X is the label, i the beginning position, j the end position and w the weight. First we populate a chart:

```
1: for j := 1 to n do
2:   for i := j − 1 to 1 by −1 do
3:     for all C in constituents such that C.i=i and C.j=j do
4:       cell_{i,j}.append(C)
5:     end for
6:   end for
7: end for
```

Then we process the chart to get the best tree, using the CYK.

```
1: n:= length of the sentence
2: for j := 1 to n do
3:   for i := j − 1 to 1 by −1 do
4:     for k := i to j − 1 do
5:       for all C ∈ cell_{k+1,j}, B ∈ cell_{i,k} and A ∈ cell_{i,j} do
6:         if not((C = X_0 − ... X_n) and (B ≠ X_n or A ≠ X_0 − ... X_{n−1}))
           then
7:           if weight(A)+totalWeight(i, B, k)+totalWeight(k+1, C, j) >
             totalWeight(i, A, j) then
8:             totalWeight(i, A, j) := weight(A)+totalWeight(i, B, k)+
               totalWeight(k + 1, C, j)
```

```
9:                backpointer(i, A, j) := (k, B, C) { to keep track of the best
                  parse tree rooted at A}
10:             end if
11:           end if
12:         end for
13:       end for
14:     end for
15: end for
16: return totalWeight(1, TOP, n) {to form the best parse tree, follow backp(1, TOP, n)}
```

$weight(A)$ returns the weight of the constituent A and $totalWeight(i, X, j)$ returns the total weight of the tree stored in $\text{cell}_{i,j}$ with a root of label X.

Appendix B

Results on Entirely Correct Trees

The first column describes the experiment and the second one gives the percentage of entirely correct trees.

Functional Parser (Baseline)	27.7
Whole Constituents, Vote Weight	21.9
Whole Constituents, Sum Weight	21.0
Whole Constituents, Occurrence Weight	21.3
Simple Experiment with Recombined Parentheses, Sum Weight	9.7
Simple Experiment with Recombined Parentheses, Occurrence Weight	9.5
Non-Empty Intersection, Sum Weight	18.1
Non-Empty Intersection, Occurrence Weight	18.5
Voted by Few, vote=1, Occ. Weight	21.3
Voted by Few, vote=2, Occ. Weight	21.3
Voted by Few, vote<3, Occ. Weight	21.3
Voted by Few, vote=3, Occ. Weight	9.8
Voted by Few and non-empty, vote=1, Occ. Weight	21.3
Voted by Few and non-empty, vote=2, Occ. Weight	21.3
Voted by Few and non-empty, vote<3, Occ. Weight	21.3
Common Spine, Sum Weight	19.3
Common Spine, Occ. Weight	19.5
Whole Constituents, Threshold	28.3
All Recombined, Threshold	9.9
Non-Empty, Threshold	19.6
Infinite Weight, All Recombined	10.3
Infinite Weight, Non-Empty	19.3
Infinite Weight, Threshold=4, All Recombined	11.0
Infinite Weight, Threshold=6, All Recombined	13.6

References

Chih-Chung Chang and Chih-Jen Lin, 2001. *LIBSVM: a library for support vector machines.*

C. Chelba and F. Jelinek. 1999. Structured language modeling for speech recognition (extended project abstract). In *Proceedings of NLDB*, Klagenfurt, Austria.

Kenneth Church and Ramesh Patil. 1982. Coping with syntactic ambiguity or how to put the block in the box on the table. *Comput. Linguist.*, 8(3-4):139–149.

T. G. Dietterich. 2000. Ensemble methods in machine learning. In *First International Workshop on Multiple Classifier Systems, Lecture Notes in Computer Science*, pages 1–15, New York: Springer Verlag. J. Kittler and F. Roli.

Dmitriy Fradkin and Ilya Muchnik. 2006. Support vector machines.

J. Henderson and E. Brill. 1999. Exploiting diversity in natural language processing: combining parsers.

J. Henderson. 2003. Inducing history representations for broad coverage statistical parsing.

C. W. Hsu, C. C. Chang, and C. J. Lin. 2003. A practical guide to support vector classification. Technical report, Taipei.

Peter Lane and James Henderson. 2001. Incremental syntactic parsing of natural language corpora with simple synchrony networks. *Knowledge and Data Engineering*, 13(2):219–231.

Yang (1) Liu, Qun Liu, and Shouxun Lin. 2006. Tree-to-string alignment template for statistical machine translation. In *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics*, pages 609–616, Sydney, Australia, July. Association for Computational Linguistics.

Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. 1994. Building a large annotated corpus of english: The penn treebank. *Computational Linguistics*, 19(2):313–330.

Paola Merlo and Gabriele Musillo. 2005. Accurate function parsing. In *Proceedings of Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing*, pages 620–627, Vancouver, British Columbia, Canada, October.

Gabriele Musillo and Paola Merlo. 2005. Lexical and structural biases for function parsing. In *Proceedings of the Ninth International Workshop on Parsing Technology*, pages 83–92, Vancouver, British Columbia, October. Association for Computational Linguistics.

Gabriele Musillo and Paola Merlo. 2006a. Accurate parsing of the proposition bank. In *Proceedings of the Human Language Technology Conference of the North American Chapter of the ACL*, pages 101–104, New York, USA, June. Association for Computational Linguistics.

Gabriele Musillo and Paola Merlo. 2006b. Robust parsing for the proposition bank. In *EACL'06 Workshop: Robust Methods in Analysis of Natural Language Data*.

Martha Palmer, Daniel Gildea, and Paul Kingsbury. 2005. The proposition bank: An annotated corpus of semantic roles. *Computational Linguistics*, 31(1):71–106, March.

Kenji Sagae and Alon Lavie. 2006. Parser combination by reparsing. In *Proceedings of the Human Language Technology Conference of the NAACL, Companion Volume: Short Papers*, pages 129–132, New York City, USA, June. Association for Computational Linguistics.

S. Sekine and M. J. Collins. 2005. Evalb, http://nlp.cs.nyu.edu/evalb/.

Mihai Surdeanu and Jordi Turmo. 2005. Semantic role labeling using complete syntactic analysis. In *Proceedings of the Ninth Conference on Computational Natural Language Learning (CoNLL-2005)*, pages 221–224, Ann Arbor, Michigan, June. Association for Computational Linguistics.

A. Taylor, M. Marcus, and B. Santorini. 2003. The penn treebank: an overview.