CONTENTS

Chapter 1

PART I: CORE TECHNOLOGIES, CHAPTER ON SYNTAX

CHAPTER OVERVIEW

- Parsing Natural Language
- Treebanks: a data-driven approach to syntax
- Syntax Analysis using Dependency Graphs and Phrase Structure Trees
- Modeling Ambiguity with Statistical Parsing
- Parsing Algorithms
- Multilingual Issues: Tokenization, Word Segmentation and Morphology

Parsing uncovers the hidden structure of linguistic input. In many applications involving natural language, the underlying predicate-argument structure of sentences can be useful. The syntactic analysis of language provides a means to explicitly discover the various predicate-argument dependencies that may exist in a sentence. In natural language processing, a syntactic analysis can vary from being very low-level, such as simply tagging each word in the sentence with a part of speech, or very high level, such as recovering a structural analysis that identifies the dependency between each predicate in the sentence and its explicit and implicit arguments. The major bottleneck in parsing natural language is the fact that ambiguity is so pervasive. In syntactic parsing, ambiguity is a particularly difficult problem since the most plausible analysis has to be chosen from an exponentially large number of alternative analyses. From tagging to full parsing, algorithms have to be carefully chosen that can handle such ambiguity. This chapter explores syntactic analysis methods from tagging to full parsing and the use of supervised machine learning to deal with ambiguity.

1.1 Parsing Natural Language

In a text to speech application input sentences are to be converted to a spoken output that should sound like it was spoken by a native speaker of the language. Consider the following pair of sentences (imagine them spoken rather than written[1]):

1. He wanted to go for a drive in movie .

2. He wanted to go for a drive in the country .

There is a natural pause between the words 'drive' and 'in' in sentence 2 which reflects an underlying hidden structure to the sentence. Parsing can provide a structural description that identifies such a break in the intonation. A simpler case occurs in the following sentence:

3. The cat who lives dangerously had nine lives .

In this case, a text to speech system needs to know that the first instance of the word 'lives' is a verb, and the second instance is a noun before it can begin to produce the natural intonation for this sentence. This is an instance of the part of speech tagging problem where each word in the sentence is assigned a most likely part of speech. These examples come from the open-source Festival text to speech system[2], which uses parsing to disambiguate these cases.

Another motivation for parsing comes from the natural language task of summarization, in which several documents about the same topic should be condensed down to a small digest of information typically limited in size to a 100 or 250 words.

[1] When written "drive in" would probably be hyphenated in the second utterance.
[2] URL for festival.

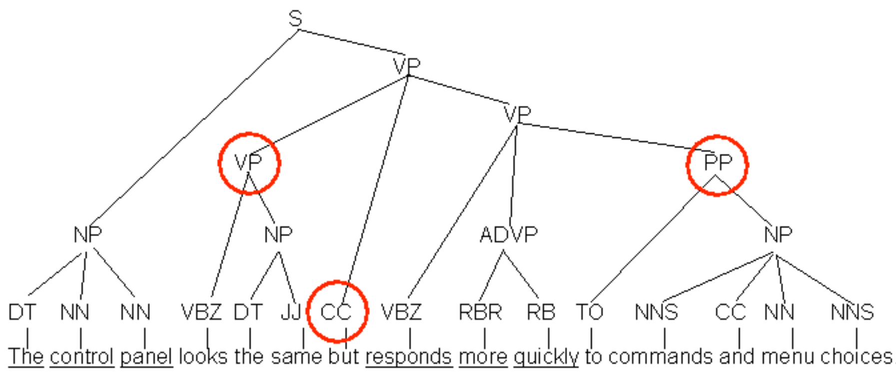


Figure 1.1. Parser output for sentence 4. Deleting the circled constituents 'VP', 'CC' and 'PP' results in retention of the underlined words which gives a shorter fluent sentence 5. From *http://www-personal.umich.edu/∼balazer/sc/* (to be replaced with figure based on Knight & Marcu).

Such a summary may be in response to a question that is answered (perhaps in different ways) in the set of documents. In this case, a useful sub-task is to compress an individual sentence so that only the relevant portions of a sentence is included in the summary (Knight and Marcu, 2001). This allows the summary to be concise, informative and fluent. For example, we may want to compress sentence 4 to a shorter sentence 5.

4. The control panel looks the same but responds more quickly to commands and menu choices .

5. The control panel responds more quickly .

An elegant way to approach this task is to first parse the sentence in order to find the various constituents: where we recursively partition the words in the sentence into individual phrases such as a verb phrase 'VP' or a noun phrase 'NP'. The output of the parser for the input sentence 4 is shown in Figure 1.1. The parse tree produced by the parser can now be edited using a compression model that is aware of constituents and few choice constituent deletions can produce a fluent compressed version of the original sentence.

Another example is the paraphrasing of text (Callison-Burch, 2007). In the sentence fragment 6 the capitalized phrase 'EUROPEAN COUNTRIES' can be replaced with other phrases without changing the essential meaning of the sentence. A few examples of replacement phrases are shown in italics in sentence fragments 7 to 11. This kind of replacement cannot simply rely on substitution of arbitrary words in the sentence since such an approach can lead to incoherent and disfluent paraphrases. Paraphrasing models build on top on parsers to identify target constituents to replace, and also to find appropriate replacement phrases that can substitute for the original phrase. Paraphrases of this type have been shown to be useful in applications such as statistical machine translation.

6. open borders imply increasing racial fragmentation in *EUROPEAN COUNTRIES* .
7. open borders imply increasing racial fragmentation in *the countries of europe* .
8. open borders imply increasing racial fragmentation in *european states* .
9. open borders imply increasing racial fragmentation in *europe* .
10. open borders imply increasing racial fragmentation in *european nations* .
11. open borders imply increasing racial fragmentation in *the european countries* .

In contemporary natural language processing, syntactic parsers are routinely used in many applications including but not limited to: statistical machine translation (Galley et al. 2006), information extraction from text collections (Miller et al. 1999), language summarization (McKeown et al. 2002), language generation (from databases) (Barzilay 2005), error correction in text, knowledge acquisition from language (e.g. discovering semantic classes or "x IS-A y" relationships) (Pantel and Lin, 2005), in speech recognition systems as language models (a language model assigns a probability to a candidate output sentence – syntax is useful in particular for disfluent or error-prone speech input) (Johnson, 2007), dialog systems (Rudnicky, 2000), text to speech systems (Festival). Parsers have been written for a large number of languages around the world, and are an essential component in many kinds of multilingual processing tasks.

1.2 Treebanks: a data driven approach to syntax

Parsing recovers information that is not explicit in the input sentence. This implies that a parser requires some knowledge in addition to the input sentence about the kind of syntactic analysis that should be produced as output. One method to provide such knowledge to the parser is to write down a grammar of the language — a set of rules of syntactic analysis. For instance, one might write down the rules of syntax as a context-free grammar (CFG). In the rest of this chapter we assume some familiarity with context-free grammars – please refer to Sipser for a good introduction to the notion of a formal grammar and CFGs in particular.

The following CFG (written in a simple Backus-Naur form) represents a simple grammar of transitive verbs in English, verbs (V) that have a subject and object noun phrase (NP), plus modifiers of verb phrases (VP) in the form of prepositional phrases (PP).

```
S -> NP VP
NP -> 'John' | 'pockets' | D N | NP PP
VP -> V NP | VP PP
V -> 'bought'
D -> 'a'
N -> 'shirt'
PP -> P NP
P -> 'with'
```

Natural language grammars typically have the words w as terminal symbols in the CFG and they are generated by rules of type $X \rightarrow w$ where X is the part of speech for the word w. For example in the above CFG, the rule `V -> 'saw'` has the part of speech symbol V generating the verb 'saw'. Such non-terminals are called part-of-speech tags or pre-terminals. The above CFG can produce a syntax analysis of a sentence like 'John bought a shirt with pockets' with S as the start symbol of the grammar. Parsing the sentence with the CFG rules gives us two possible derivations for this sentence. In one parse, pockets are a kind of currency which can be used to buy a shirt, and the other parse, which is the more plausible one, John is purchasing a kind of shirt that has pockets.

```
(S (NP John)
   (VP (VP (V bought)
           (NP (D a)
               (N shirt)))
       (PP (P with)
           (NP pockets))))
```

```
(S (NP John)
   (VP (V bought)
       (NP (NP (D a)
               (N shirt))
           (PP (P with)
               (NP pockets)))))
```

However, writing down a CFG for the syntactic analysis of natural language is problematic. Unlike a programming language, natural language is far too complex to simply list all the syntactic rules in terms of a CFG. A simple list of rules does not consider interactions between different components in the grammar. We could extend this grammar to include other types of verbs, and other syntactic constructions, but listing all possible syntactic constructions in a language is a difficult task. In addition it is difficult to exhaustively list lexical properties of words, for instance, listing all the grammar rules in which a particular word can be a participant. This is a typical knowledge acquisition problem.

Apart from this knowledge acquisition problem, there is another less apparent problem: it turns out that the rules interact with each other in combinatorially explosive ways. Consider a simple CFG that provides a syntactic analysis of noun phrases as a binary branching tree:

```
N -> N N
N -> 'natural' | 'language' | 'processing' | 'book'
```

Recursive rules produce ambiguity: with N as the start symbol, for the input 'natural' there is one parse tree `(N natural)`; for the input 'natural language' we use the recursive rule once and obtain one parse tree `(N (N natural) (N language))`; for the input 'natural language processing' we use the recursive rule twice in each parse and there are two ambiguous parses:

```
(N (N (N natural)                 (N (N natural)
      (N language))                  (N (N language)
   (N processing))                      (N processing)))
```

Note that the ambiguity in the syntactic analysis reflects a real ambiguity: is it a processing of natural language, or is it a natural way to do language processing? So this issue cannot be resolved by changing the formalism in which the rules are written – e.g. by using finite-state automata which can be deterministic but cannot simultaneously model both meanings in a single grammar. Any system of writing down syntactic rules should represent this ambiguity. However, by using the recursive rule 3 times we get 5 parses for 'natural language processing book' and for longer and longer input noun phrases, using the recursive rule 4 times we get 14 parses, using it 5 times we get 42 parses, using it 6 times we get 132 parses. In fact, for CFGs it can be proved that the number of parses obtained by using the recursive rule n times is the Catalan number of n:

$$Cat(n) = \frac{1}{n+1} \begin{pmatrix} 2n \\ n \end{pmatrix}$$

This occurs not only for coordinate structures such as the noun phrase grammar, but also when you have recursive rules to deal with modifiers such as the recursive rule for prepositional phrase modification `VP -> VP PP` in the first CFG in this section. In fact, the ambiguity of PP modification is not independent of the ambiguity of coordination: in a sentence with both types of ambiguity, the total number of parses is the cross product of the parses from each sub-grammar. This poses a serious computational problem for parsers. For an input with n words the number of possible parses is exponential in n.

For most natural language tasks we do not wish to explore this entire space of ambiguity. For example, for the input 'natural language processing book' only one out of the five parses obtained using the CFG above is intuitively correct (corresponding to a book about the processing of natural language):

```
(N (N (N (N natural)
         (N language))
      (N processing))
   (N book))
```

This is a second knowledge acquisition problem – not only do we need to know the syntactic rules for a particular language, but we also need to know which analysis is the most plausible for a given input sentence. The construction of a *treebank* is a data driven approach to syntax analysis allows us to address both of these knowledge acquisition bottlenecks in one stroke.

A treebank is simply a collection of sentences (also called a corpus of text), where each sentence is provided a complete syntax analysis. The syntactic analysis for each sentence has been judged by a human expert as the most plausible analysis for that sentence. A lot of care is taken during the human annotation process to ensure that a consistent treatment is provided across the treebank for related grammatical phenomena. A style-book or annotation guideline is typically written before the annotation process in order to ensure a consistent scheme of annotation throughout the treebank.

There is no set of syntactic rules or linguistic grammar explicitly provided by a treebank, and typically there is no list of syntactic constructions provided explicitly in a treebank. In fact, no exhaustive set of rules is even assumed to exist, even though assumptions about syntax are implicit in a treebank. A detailed set of assumptions about syntax is typically used as an *annotation guideline* to help the human experts produce the single most plausible syntactic analysis for each sentence in the corpus. The consistency of syntax analysis in a treebank is measured using inter-annotator agreement by having approximately 10% overlapped material annotated by more than one annotator.

Treebanks provide a solution to the two kinds of knowledge acquisition bottlenecks we discussed above. Treebanks provide annotations of syntactic structure for a large sample of sentences. We can use supervised machine learning methods in order to train a parser to produce a syntactic analysis for input sentences by generalizing appropriately from the training data extracted from the treebank.

Treebanks solve the first knowledge acquisition problem of finding the grammar underlying the syntax analysis since the syntactic analysis is directly given instead of a grammar. In fact, the parser does not necessarily need any explicit grammar rules as long as it can faithfully produce a syntax analysis for an input sentence – although the information used by the trained parser can be said to represent a set of implicit grammar rules. (Nivre, 2005) discusses in further detail this subtle difference between parsing using a grammar and parsing a text using data-driven methods which may or may not be grammar-based.

Treebanks solve the second knowledge acquisition problem as well. Since each sentence in a treebank has been given its most plausible syntactic analysis, supervised machine learning methods can be used to learn a scoring function over all possible syntax analyses. A statistical parser trained on the treebank tries to mimic the human annotation decisions by using indicators from the input and previous decisions made in the parser itself to learn such a scoring function. For a given sentence unseen in the training data, a statistical parser can use this scoring function to return the syntax analysis that has the highest score – which is taken to be the most plausible analysis for that sentence. The scoring function can also be used to produce the k-best syntax analyses for a sentence.

There are two main approaches to syntax analysis which are used to construct treebanks: dependency graphs and phrase structure trees. These two representations are very closely related to each other – and under some assumptions one representation can be converted to another. Dependency analysis is typically favored for languages such as Czech, Turkish, etc. that have free(er) word order, where the arguments of a predicate are often seen in different ordering in the sentence, while phrase-structure analysis is often used to provide additional information about long-distance dependencies and mostly in languages like English, French, etc. where the word order is less flexible.

1.3 Syntax Analysis using Dependency Graphs

The main philosophy behind dependency graphs (Tesniere, 1933; Hudson, 1984) is to make minimal assumptions about syntactic structure and in particular to avoid any annotation of hidden structure such as, for example, using empty elements as placeholders to represent missing or displaced arguments of predicates, or any unnecessary hierarchical structure. The words in the input sentence are treated as the only vertices in the graph which are linked together by directed arcs which represent syntactic dependencies. The CoNLL 2007 shared task on dependency parsing (Nivre et al. 2007) provides the following definition of a *dependency graph*:

> In dependency-based syntactic parsing, the task is to derive a syntactic structure for an input sentence by identifying the syntactic *head* of each word in the sentence. This defines a *dependency graph*, where the nodes are the words of the input sentence and the arcs are the binary relations from head to dependent. Often, but not always, it is assumed that all words except one have a syntactic head, which means that the graph will be a tree with the single independent node as the root. In *labeled* dependency parsing, we additionally require the parser to assign a specific type (or label) to each dependency relation holding between head word and dependent word.

As in the above definition, we will restrict ourselves to dependency tree analyses, where each word depends on exactly one parent, either another word or a dummy root symbol. By convention, in dependency trees the 0 index is used to indicate the root symbol and the directed arcs are drawn from the head word to the dependent word. For example, Figure 1.2 shows an example of a dependency tree for a Czech sentence taken from the Prague Dependency Treebank, which is a large annotated corpus of Czech text annotated with dependency trees (Each treebank has its own annotation flavor and the Prague treebank annotates other levels of information as well, such as topic and focus structure of the sentence, but we only show the dependency tree information here).

There are many variants of dependency style syntactic analysis, but the basic textual format for a dependency tree can be written in the following form, where each dependent word specifies the head word in the sentence, and exactly one word

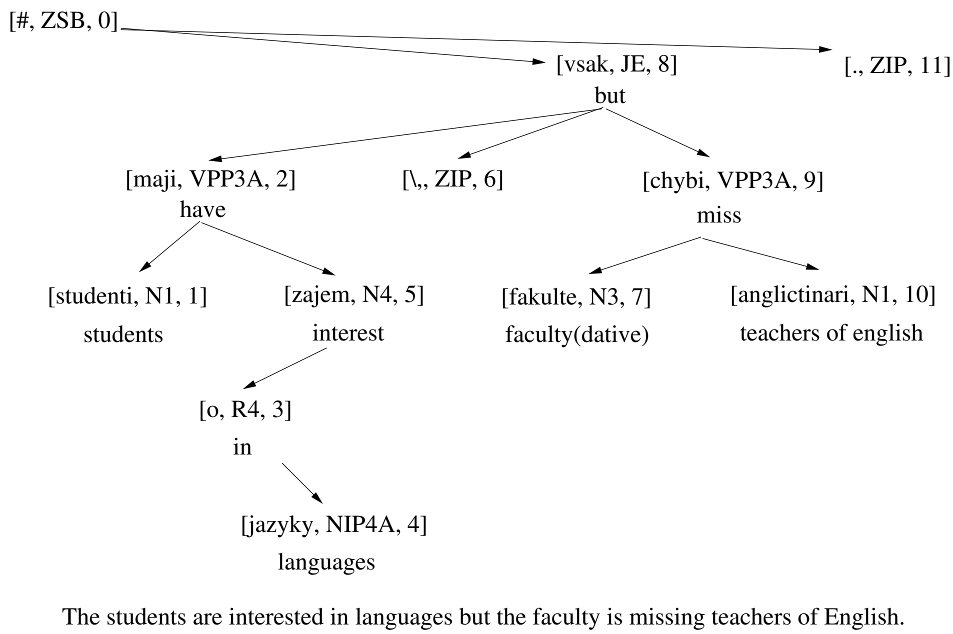


Figure 1.2. An example of a dependency graph syntax analysis for a Czech sentence taken from the Prague Dependency Treebank. Each node in the graph is a word, its part of speech and the position of the word in the sentence, e.g. [fakulte, N3, 7] is the 7th word in the sentence with part of speech tag N3 which also tells us that the word has dative case. The node [#,ZSB,0] is the root node of the dependency tree. The English equivalent is provided for each node.

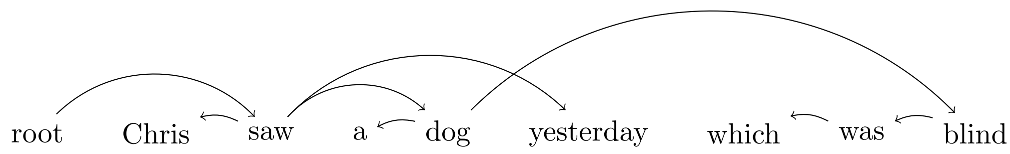


Figure 1.3. An unlabeled *non-projective* dependency tree, with a crossing dependency.

is dependent to the root of the sentence. The following shows a typical textual representation of a labeled dependency tree:

index	word	part-of-speech	head	label
1	They	PRP	2	SBJ
2	persuaded	VBD	0	ROOT
3	Mr.	NNP	4	NMOD
4	Trotter	NNP	2	IOBJ
5	to	TO	6	VMOD
6	take	VB	2	OBJ
7	it	PRP	6	OBJ
8	back	RB	6	PRT
9	.	.	2	P

An important notion in dependency analysis is the notion of *projectivity* which is a constraint imposed by the linear order of words on the dependencies between words (Gaifman 1955). A *projective dependency tree* is one where if we put the words in a linear order based on the sentence with the root symbol in the first position, the dependency arcs can be drawn above the words without any crossing dependencies. Another way to state projectivity is to say that for each word in the sentence, its descendants form a contiguous substring of the sentence. For example, Figure 1.3 shows a natural analysis of an English sentence that contains an extraposition to the right of a noun phrase modifier phrase, which as a result requires a crossing dependency. However, English has a very small number of cases in a treebank that will need such a non-projective analysis. In other languages, such Czech, Turkish, etc. . . . the number of non-projective dependencies can be much more frequent. As a percentage of the total number of dependencies, crossing dependencies even in those languages are a small percentage. However, a substantial percentage of sentences contain at least one crossing dependency making it an important issue in some languages. Table 1.1 contains a multi-lingual comparison of crossing dependencies across the languages that were part of the CoNLL 2007 shared task on dependency parsing.

Dependency graphs in treebanks do not explicitly distinguish between projective and *non-projective dependency tree* analyses. However, parsing algorithms are sometimes forced to make a distinction between projective and non-projective de-

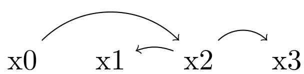
	Ar	Ba	Ca	Ch	Cz	En	Gr	Hu	It	Tu
% deps	0.4	2.9	0.1	0.0	1.9	0.3	1.1	2.9	0.5	5.5
% sents	10.1	26.2	2.9	0.0	23.2	6.7	20.3	26.4	7.4	33.3

Table 1.1. A multilingual comparison of percent of crossing dependencies and percent of sentences with non-projectivity taken from the CoNLL 2007 shared task data set. Ar = Arabic, Ba = Basque, Ca = Catalan, Ch = Chinese, Cz = Czech, En = English, Gr = Greek, Hu = Hungarian, It = Italian, Tu = Turkish. Note that in some cases the dependency trees were created by conversion via heuristic rules from an original phrase-structure tree.

pendencies. Let us examine this distinction further using context-free grammars. Note that we can set up dependency links in a context-free grammar. For example,

```
X0_2 -> X0_1* X2_1
X0_1 -> x0*
X2_1 -> X1_1 X2_2*
X1_1 -> x1*
X2_2 -> X2_3* X3_1
X2_3 -> x2*
X3_1 -> x3*
```

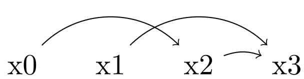
In the above CFG, the terminal symbols are `x0, x1, x2, x3` and the asterisk picks out a single symbol in the right hand side of each rule that specifies the dependency links. In this example, the dependency tree equivalent to the above CFG is as shown below:



We can show that if we can convert a dependency tree into an equivalent CFG (using the notation used above) then the dependency tree must be projective. In a CFG converted from a dependency tree we have only the following three types of rules with one type of rule to introduce the terminal symbols and two rules where Y is dependent on X or vice versa. The head word of X or Y can be traced by following the asterisk symbol.

```
Z -> X* Y
Z -> X Y*
A -> a*
```

Assume that we have a non-projective dependency tree. For example,



Converting such a dependency tree to a CFG with the asterisk notation gives us two options. Either we can capture X3 depends on X2 but fail to capture that X1 depends on X3:

```
X2_3 -> X1_1 X2_2*
X1_1 -> x1
X2_2 -> X2_1* X3_1
X2_1 -> x2
X3_1 -> x3
```

Or capture the fact that X1 depends on X3 but fail to capture that X3 depends on X2:

```
X2_3 -> X1_1 X3_2*
X1_1 -> x1
X3_2 -> X2_1 X3_1*
X2_1 -> x2
X3_1 -> x3
```

In fact, there is no CFG that can capture the non-projective dependency. Recall that projectivity can be defined as follows: for each word in the sentence, its descendants form a contiguous substring of the sentence. Thus, non-projectivity can be defined as follows: a non-projective dependency means that there is a word in the sentence (or equivalently a non-terminal in the CFG created from the dependency tree) such that its descendants do not form a contiguous substring of the sentence. Put another way there is a nonterminal Z such that Z derives spans (x_i, x_k) and (x_{k+p}, x_j) for some $p > 0$. This means there must be a rule $Z \rightarrow PQ$ where P derives (x_1, x_k) and Q derives (x_{k+p}, x_j). However, by definition, this is only valid in CFGs if $k = 0$ since P and Q *must* be contiguous substrings. Hence no dependency tree with non-projective dependencies can be converted into an equivalent (asterisk-marked) CFG.

This gives a useful characterization of projective dependencies in terms of context-free grammar derivations. If we want a dependency parser to only produce projective dependencies, we can implicitly create an equivalent CFG that will ignore all non-projective dependencies. We will explore this further when we discuss parsing algorithms.

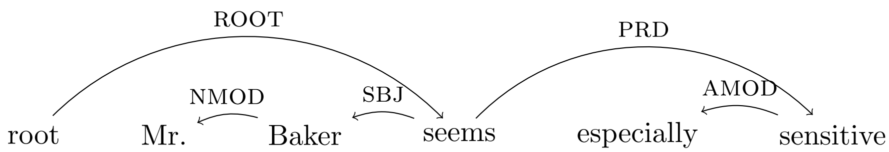
1.4 Syntax Analysis using Phrase Structure Trees

A phrase structure syntax analysis of a sentence derives from the traditional sentence diagrams which partition a sentence into constituents and larger constituents are formed by merging smaller ones. Phrase structure analysis also typically incorporate ideas from generative grammar (from Linguistics) in order to deal with displaced constituents or apparent long distance relationships between heads and constituents. A phrase structure tree can be viewed as implicitly having a predicate-argument structure associated with it. For example, the following phrase structure analysis of the sentence *Mr. Baker seems especially sensitive* taken from the Penn Treebank shows that the subject of the sentence is marked with the -SBJ marker and that the predicate of the sentence is also marked with the -PRD marker. The underlying predicate-argument structure is shown below the tree using an informal notation that captures the information implied by the phrase-structure tree.

```
(S (NP-SBJ (NNP Mr.)
           (NNP Baker))
   (VP (VBZ seems)
       (ADJP-PRD (RB especially)
                 (JJ sensitive))))
```

```
Predicate-argument structure:
seems((especially(sensitive))(Mr. Baker))
```

The same sentence gets the following dependency tree analysis. Note how some of the information from the bracketing labels from the phrase-structure analysis gets mapped onto the labeled arcs of the dependency analysis. Typically, dependency analysis would not link the subject with the predicate directly because it would create an inconvenient crossing dependency with the dependency between *seems* and the *root* symbol.



In order to explain some details of phrase-structure analysis in treebanks, we use some examples of syntax analysis that show how null elements (constituents with no yield) are used in order to localize certain predicate-argument dependencies in the tree structure. The examples are taken from the ARPA 1994 paper about the Penn Treebank. The part of speech tags for the words have been omitted to simplify the tree.

In the first example, we see that an NP dominates a trace *T* which is a null element, the same as an epsilon symbol in formal language theory, having no yield in the input. This empty trace has an index (here it is 1, but the actual value is not important) and associated with the WHNP constituent with the same index. This co-indexation allows us to infer the predicate-argument structure shown below the tree.

```
(SBARQ (WHNP-1 What)
       (SQ is (NP-SBJ Tim)
              (VP eating (NP *T*-1)))
       ?)

Predicate-argument structure:
eat(Tim, what)
```

In the second example, the subject of the sentence *The ball* is actually not the logical subject of the predicate which has been displaced due to the passive construction. The logical subject of the sentence *Chris* is marked as -LGS enabling the recovery of the predicate argument structure for this sentence.

```
(S (NP-SBJ-1 The ball)
   (VP was (VP thrown)
         (NP *-1)
         (PP by (NP-LGS Chris))))
```

```
Predicate-argument structure:
throw(Chris, the ball)
```

The third example shows that different syntactic phenomena are often combined in the corpus and both the analyses above are combined to provide the predicate argument structure in such cases.

```
(SBARQ (WHNP-1 Who)
       (SQ was (NP-SBJ-2 *T*-1)
               (VP believed (S (NP-SBJ-3 *-2)
                               (VP to (VP have
                                        (VP been
                                          (VP shot
                                            (NP *-3))))))))
       ?)

Predicate-argument structure:
believe(*someone*, shoot(*someone*, who))
```

The fourth pair of examples shows how null elements are used to annotate the presence of a subject for a predicate even if it is not explicit in the sentence. In the first case, the phrase structure annotation in the treebank marks the missing subject for *take back* as the object of the verb *persuaded*.

```
(S (NP-SBJ (PRP They))
   (VP (VP (VBD persuaded)
           (NP-1 (NNP Mr.)
                 (NNP Trotter))
           (S (NP-SBJ (-NONE- *-1))
              (VP (TO to)
                  (VP (VB take)
                      (NP (PRP it))
                      (PRT (RB back))))))))

Predicate argument structure:
persuade(they, Mr. Trotter, take_back(Mr. Trotter, it))
```

In the first case, the phrase structure annotation in the treebank marks the missing subject for *take back* as the subject of the verb *promised*.

```
(S (NP-SBJ-1 (PRP They))
   (VP (VP (VBD promised)
           (NP (NNP Mr.)
                 (NNP Trotter))
           (S (NP-SBJ (-NONE- *-1))
              (VP (TO to)
                  (VP (VB take)
```

```
                    (NP (PRP it))
                    (PRT (RB back))))))))

Predicate argument structure:
promise(they, Mr. Trotter, take_back(they, it))
```

The dependency analysis for *persuaded* and *promised* do not make such a distinction. The dependency analysis for the two sentences in the pair of examples above would be identical, as shown below.

```
1 They      PRP 2 SBJ                  1 They     PRP 2 SBJ
2 persuaded VBD 0 ROOT                 2 promised VBD 0 ROOT
3 Mr.       NNP 4 NMOD                 3 Mr.      NNP 4 NMOD
4 Trotter   NNP 2 IOBJ                 4 Trotter  NNP 2 IOBJ
5 to        TO  6 VMOD                 5 to       TO  6 VMOD
6 take      VB  2 OBJ                  6 take     VB  2 OBJ
7 it        PRP 6 OBJ                  7 it       PRP 6 OBJ
8 back      RB  6 PRT                  8 back     RB  6 PRT
9 .         .   2 P                    9 .        .   2 P
```

However, while pointing out these differences in annotation philosophy between dependency and phrase-structure treebanks, it is important to note that most parsers that are trained using phrase-structure treebanks typically ignore these differences. The rich annotation of logical subjects, null elements, etc. ... are all but ignored in modern statistical parsers. There has been some interest in recovering the underlying predicate-argument structure using these rich annotations, but by and large parsers trained on phrase-structure treebanks still only report performance on what is observed at the surface level such as constituents found correctly, or dependencies recovered correctly. Mention work on null element restoration here and work by Stephen Clark et al. in EMNLP 2009.

In different treebanks for the same language, or treebanks for different languages, there might be many differences in the phrase structure annotation. The differences could be in the choice of symbols and what they represent. In the following example from the Chinese treebank, the symbol IP is used instead of S which reflects a move from the English Penn Treebank's predominantly transformational grammar–based phrase structure to Government-Binding (GB) based phrase structures. The differences could also be related to specific syntactic construction. In the following example, the possessive marker 的 is given a particular analysis which results in a fairly complex structural analysis with several null elements for 新 的, with a null WHNP even though Chinese has no relative pronouns. This structure is motivated by the perceived need for a uniform and consistent phrase structure for clauses and clause-like constituents throughout the treebank. Such differences mean that a phrase-structure parser developed initially for English parsing and trained on the English treebank may not be easily portable to another language even though a phrase-structure treebank exists for that language. (Levy and Manning, 2002) discuss the many challenges in taking a context-free grammar based parser initially developed for English parsing and adapting it to Chinese parsing by training on the Chinese phrase-structure treebank.

```
(IP (NP-SBJ (NP (NN 结售/settlement and sale)
                (NN 制度/system))
            (CC 和/and)
            (NP (CP (WHNP-2 (-NONE- *OP*))
                    (CP (IP (NP-SBJ (-NONE- *T*-2))
                            (VP (VA 新/new)))
                        (DEC 的)))
                (NP (NN 核销/verification and cancellation)
                    (NN 制度/system))))
    (VP (PP-LOC (P 在/in)
                (NP-PN (NR 西藏/Tibet)))
        (ADVP (AD 全面/fully))
        (VP (VV 实施/operating))))

English translation:
A (foreign exchange) settlement and sale system and a verification and
cancellation system that is newly created is fully operational in Tibet.
```

1.5 Statistical Parsing

In this section we focus on the modeling aspect of parsing: how to design features and ways to resolve ambiguity in parsing. Using these models to parse efficiently will be covered in the next section when we cover parsing algorithms. The algorithms from that section can efficiently use the models described in this section to find the highest scoring parse tree or dependency analysis.

1.5.1 Probabilistic Context-free Grammars

Consider the ambiguity problem we discussed earlier, where we would like to choose between the following ambiguous parses for the sentence 'John bought a shirt with pockets'.

```
(S (NP John)                         (S (NP John)
   (VP (VP (V bought)                   (VP (V bought)
           (NP (D a)                        (NP (NP (D a)
               (N shirt)))                          (N shirt))
       (PP (P with)                             (PP (P with)
           (NP pockets))))                          (NP pockets)))))
```

We want to provide a model that would match the intuition that the second tree above is preferred over the first. The parses themselves can be thought of as ambiguous (leftmost or rightmost) derivations of the following CFG.

```
S -> NP VP
```

```
NP -> 'John' | 'pockets' | D N | NP PP
VP -> V NP | VP PP
V -> 'bought'
D -> 'a'
N -> 'shirt'
PP -> P NP
P -> 'with'
```

We can add scores or probabilities to the rules in this CFG in order to provide a score or probability for each derivation. The probability of a derivation is simply the sum of scores or product of probabilities of all the CFG rules used in that derivation. Since scores can be viewed simply as log probabilities we will use the term probabilistic context-free grammar (PCFG) when scores or probabilities are assigned to CFG rules. In order to make sure that the probability of the set of trees generated by a PCFG is well-defined, we assign probabilities to the CFG rules such that for a rule $N \rightarrow \alpha$ the probability is $P(N \rightarrow \alpha \mid N)$, that is each rule probability is conditioned on the left-hand side of the rule. This means that during the context-free expansion of a non-terminal – the probability is distributed among all the expansions of the non-terminals. In other words,

$$1 = \sum_{\alpha} P(N \rightarrow alpha)$$

So in the above example we can assign probabilities to rules in the CFG to obtain the result we want – the more plausible parse gets the higher probability.

```
S -> NP VP (1.0)
NP -> 'John' (0.1) | 'pockets' (0.1) | D N (0.3) | NP PP (0.5)
VP -> V NP (0.9) | VP PP (0.1)
V -> 'bought' (1.0)
D -> 'a' (1.0)
N -> 'shirt' (1.0)
PP -> P NP (1.0)
P -> 'with' (1.0)
```

From these rule probabilities, the only deciding factor for deciding among the two parses for 'John bought a shirt with pockets' is the two rules `NP -> NP PP` and `VP -> VP PP` since all the other rules in one parse also occur in the other. Since the probability for `NP -> NP PP` has been set higher in the PCFG above, the most plausible analysis gets the higher probability.

However the definition of PCFGs meant that various other rule probabilities had to be adjusted in order to obtain the right scoring of parses. Also, the independence assumptions in PCFG, which are dictated by the underlying CFG, often leads to bad models which cannot use information vital to the decision of rule scores which lead to high scoring plausible parses. We would like to model such ambiguities using arbitrary "features" of the parse tree. Discriminative methods provide us with such a class of models.

1.5.2 Global Linear Models

Michael Collins [1] provides a common framework called *global linear models* to describe various discriminative approaches to learning for parsing (and also chunking or tagging). Let $\mathbf{x}$ be a set of inputs, and $\mathbf{y}$ be a set of possible outputs which can be a sequence of part of speech tags, or a parse tree or a dependency analysis.

- Each $x \in \mathbf{x}$ and $y \in \mathbf{y}$ is mapped to a d-dimensional feature vector $\Phi(x,y)$, with each dimension being a real number, summarizing partial information contained in (x,y).

- A weight parameter vector $\mathbf{w} \in \Re^d$ assigns a weight to each feature in $\Phi(x,y)$, representing the importance of that feature. The value of $\Phi(x,y) \cdot \mathbf{w}$ is the score of (x,y). The higher the score, the more plausible it is that y is the output for x.

- The function $GEN(x)$ generates the set of possible outputs y for a given x.

Having $\Phi(x,y)$, $\mathbf{w}$, and $GEN(x)$ specified, we would like to choose the highest scoring candidate y^* from $GEN(x)$ as the most plausible output. That is,

$$F(x) = \operatorname*{argmax}_{y \in GEN(x)} p(y \mid x, \mathbf{w})$$

where $F(x)$ returns the highest scoring output y^* from $GEN(x)$. A *conditional random field* (CRF) [2] defines the conditional probability as a linear score for each candidate y and a *global* normalization term:

$$\log p(y \mid x, \mathbf{w}) = \Phi(x, y) \cdot \mathbf{w} - \log \sum_{y' \in GEN(x)} \exp(\Phi(x, y') \cdot \mathbf{w})$$

In our experiments we find that a simpler global linear model that ignores the normalization term is faster to train and provides comparable accuracy.

$$F(x) = \operatorname*{argmax}_{y \in GEN(x)} \Phi(x, y) \cdot \mathbf{w}$$

For this model, we learn the weight vector from labeled data using the perceptron algorithm [1]. A global linear model is global is two ways: it uses features that are defined over the entire sequence, *and* the parameter estimation methods are explicitly related to errors over the entire sequence.

A perceptron [3] is a single-layered neural network. It is trained using online learning, that is, processing examples one at a time, during which it adjusts a weight parameter vector that can then be applied on input data to produce the corresponding output. The weight adjustment process awards features appearing in the truth and penalizes features not contained in the truth. After the update,

Inputs: Training Data $\langle (x_1, y_1), \ldots, (x_m, y_m) \rangle$; number of iterations T
Initialization: Set $\mathbf{w} = \mathbf{0}$
Algorithm:
  for $t = 1, \ldots, T$ **do**
    for $i = 1, \ldots, m$ **do**
      Calculate $y_i^{'}$, where $y_i^{'} = \operatorname*{argmax}_{y \in GEN(x)} \Phi(x_i, y) \cdot \mathbf{w}$
      if $y_i^{'} \neq y_i$ **then**
        $\mathbf{w} = \mathbf{w} + \Phi(x_i, y_i) - \Phi(x_i, y_i^{'})$
      end if
    end for
  end for
Output: The updated weight parameter vector $\mathbf{w}$

Figure 1.4. The original perceptron learning algorithm

the perceptron ensures that the current weight parameter vector is able to correctly classify the present training example.

Suppose we have m examples in the training set. The original perceptron learning algorithm [3] is shown in Figure 1.4.

The weight parameter vector $\mathbf{w}$ is initialized to $\mathbf{0}$. Then the algorithm iterates through those m training examples. For each example x, it generates a set of candidates $GEN(x)$, and picks the most plausible candidate, which has the highest score according to the current $\mathbf{w}$. After that, the algorithm compares the selected candidate with the truth, and if they are different from each other, $\mathbf{w}$ is updated by increasing the weight values for features appearing in the truth and by decreasing the weight values for features appearing in this top candidate. If the training data is linearly separable, meaning that it can be discriminated by a function which is a linear combination of features, the learning is proven to converge in a finite number of iterations [4].

This original perceptron learning algorithm is simple to understand and to analyze. However, the incremental weight updating suffers from over-fitting, which tends to classify the training data better, at the cost of classifying the unseen data worse. Also, the algorithm is not capable to deal with training data that is linearly inseparable.

Freund and Schapire [4] proposed a variant of the perceptron learning approach — the voted perceptron algorithm. Instead of storing and updating parameter values inside one weight vector, its learning process keeps track of all intermediate weight vectors, and these intermediate vectors are used in the classification phase to vote for the answer. The intuition is that good prediction vectors tend to survive for a long time and thus have larger weight in the vote. Figure 1.5 shows the voted perceptron training and prediction phases from [4], with slightly modified representation.

The voted perceptron keeps a count c_i to record the number of times a particular weight parameter vector $(\mathbf{w}_i, c_i)$ survives in the training. For a training example, if its selected top candidate is different from the truth, a new count c_{i+1}, being initialized to 1, is used, and an updated weight vector $(\mathbf{w}_{i+1}, c_{i+1})$ is produced; meanwhile, the original c_i and weight vector $(\mathbf{w}_i, c_i)$ are stored.

Compared with the original perceptron, the voted perceptron is more stable, due to maintaining the list of intermediate weight vectors for voting. Nevertheless, to store those weight vectors is space inefficient. Also, the weight calculation, using all intermediate weight parameter vectors during the prediction phase, is time consuming.

The averaged perceptron algorithm [1], an approximation to the voted perceptron, on the other hand, maintains the stability of the voted perceptron algorithm, but significantly reduces space and time complexities. In an averaged version, rather than using $\mathbf{w}$, the averaged weight parameter vector γ over the m training examples is used for future predictions on unseen data:

Training Phase
Input: Training data $\langle(x_1, y_1), \ldots, (x_m, y_m)\rangle$, number of iterations T
Initialization: k = 0, $\mathbf{w}_0 = \mathbf{0}$, $c_1 = 0$
Algorithm:

```
for t = 1,...,T do
  for i = 1,...,m do
    Calculate y'_i, where y'_i = argmax_{y∈GEN(x)} Φ(x_i, y) · w_k
    if y'_i = y_i then
      c_k = c_k + 1
    else
      w_{k+1} = w_k + Φ(x_i, y_i) − Φ(x_i, y'_i)
      c_{k+1} = 1
      k = k + 1
    end if
  end for
end for
```

Output: A list of weight vectors $\langle(\mathbf{w}_1, c_1), \ldots, (\mathbf{w}_k, c_k)\rangle$

Prediction Phase
Input: The list of weight vectors $\langle(\mathbf{w}_1, c_1), \ldots, (\mathbf{w}_k, c_k)\rangle$, an unsegmented sentence x
Calculate:

$$y^* = \underset{y \in GEN(x)}{\operatorname{argmax}} \left(\sum_{i=1}^{k} c_i \Phi(x, y) \cdot \mathbf{w}_i \right)$$

Output: The voted top ranked candidate y^*

Figure 1.5. The voted perceptron algorithm

$$\gamma = \frac{1}{\mathrm{mT}} \sum_{i=1\ldots m, t=1\ldots T} \mathbf{w}^{i,t}$$

In calculating γ, an accumulating parameter vector σ is maintained and updated using $\mathbf{w}$ for each training example. After the last iteration, $\sigma/(\mathrm{mT})$ produces the final parameter vector γ. The entire algorithm is shown in Figure 1.6.

Inputs: Training Data $\langle (x_1, y_1), \ldots, (x_m, y_m) \rangle$; number of iterations T
Initialization: Set $\mathbf{w} = \mathbf{0}$, $\gamma = \mathbf{0}$, $\sigma = \mathbf{0}$
Algorithm:

```
for t = 1, ..., T do
  for i = 1, ..., m do
    Calculate y'_i, where y'_i = argmax_{y∈GEN(x)} Φ(x_i, y) · w
    if y'_i ≠ y_i then
      w = w + Φ(x_i, y_i) − Φ(x_i, y'_i)
    end if
    σ = σ + w
  end for
end for
```

Output: The averaged weight parameter vector $\gamma = \sigma/(\mathrm{mT})$

Figure 1.6. The averaged perceptron learning algorithm

When the number of features is large, it is expensive to calculate the total parameter σ for each training example. To further reduce the time complexity, Collins [5] proposed the lazy update procedure. After processing each training sentence, not all dimensions of σ are updated. Instead, an update vector τ is used to store the exact location (p,t) where each dimension of the averaged parameter vector was last updated, and only those dimensions corresponding to features appearing in the current sentence are updated. p represents the training example index where this particular feature was last updated, and t represents its corresponding iteration number. While for the last example in the final iteration, each dimension of τ is updated, no matter whether the candidate output is correct or not. Figure 1.7 shows the averaged perceptron with lazy update procedure.

In order to find the most plausible parse tree, the parser has to choose between the possible derivations each of which can be represented as a sequence of decisions. Let each derivation $D = d_1, \ldots, d_n$ which is the sequence of decisions used to build the parse tree. Then for input sentence x the output parse tree y is defined by the sequence of steps in the derivation. We can introduce a probability for each derivation:

Inputs: Training Data $\langle(x_1, y_1), \ldots, (x_m, y_m)\rangle$; number of iterations T
Initialization: Set $\mathbf{w} = \mathbf{0}$, $\gamma = \mathbf{0}$, $\sigma = \mathbf{0}$, $\tau = \mathbf{0}$
Algorithm:

```
for t = 1,..., T do
  for i = 1,..., m do
    Calculate y'_i, where y'_i = argmax_{y∈GEN(x)} Φ(x_i, y) · w
    if t ≠ T or i ≠ m then
      if y'_i ≠ y_i then
        // Update active features in the current sentence
        for each dimension s in (Φ(x_i, y_i) − Φ(x_i, y'_i)) do
          if s is a dimension in τ then
            // Include the total weight during the time
            // this feature remains inactive since last update
            σ_s = σ_s + w_s · (t · m + i − t_{τ_s} · m − i_{τ_s})
          end if
          // Also include the weight calculated from comparing y'_i with y_i
          w_s = w_s + Φ(x_i, y_i) − Φ(x_i, y'_i)
          σ_s = σ_s + Φ(x_i, y_i) − Φ(x_i, y'_i)
          // Record the location where the dimension s is updated
          τ_s = (i, t)
        end for
      end if
    else
      // To deal with the last sentence in the last iteration
      for each dimension s in τ do
        // Include the total weight during the time
        // each feature in τ remains inactive since last update
        σ_s = σ_s + w_s · (T · m + m − t_{τ_s} · m − i_{τ_s})
      end for
      // Update weights for features appearing in this last sentence
      if y'_i ≠ y_i then
        w = w + Φ(x_i, y_i) − Φ(x_i, y'_i)
        σ = σ + Φ(x_i, y_i) − Φ(x_i, y'_i)
      end if
    end if
  end for
end for
```

Output: The averaged weight parameter vector $\gamma = \sigma/(\text{mT})$

Figure 1.7. The averaged perceptron learning algorithm with lazy update procedure

$$P(x, y) = P(d_1, \ldots, d_n) = \prod_{i=1}^{n} P(d_i \mid d_1, \ldots, d_{i-1})$$

The conditioning context in the probability $P(d_i \mid d_1, \ldots, d_{i-1})$ is called the *history* and corresponds to a partially build parse tree (as defined by the derivation sequence). We make a simplifying assumption that keeps the conditioning context to a finite set by grouping the histories into equivalence classes using a function Φ.

$$P(d_1, \ldots, d_n) = \prod_{i=1}^{n} P(d_i \mid \Phi(d_1, \ldots, d_{i-1}))$$

Using Φ each history $H_i = d_1, \ldots, d_{i-1}$ for all x, y is mapped to some fixed finite set of feature functions of the history $\phi_1(H_i), \ldots, \phi_k(H_i)$. In terms of these k feature functions:

$$P(d_1, \ldots, d_n) = \prod_{i=1}^{n} P(d_i \mid \phi_1(H_i), \ldots, \phi_k(H_i))$$

1.6 Parsing Algorithms

Given an input sentence, a parser produces an output analysis of that sentence, which we now assume is the analysis that is consistent with a treebank that is used to *train* a parser. Treebank parsers do not need to have an explicit grammar, but to make the explanation of parsing algorithms simpler we first consider parsing algorithms that assume the existence of a context-free grammar.

Consider the following simple CFG G that can be used to derive strings such as *a and b or c* from the start symbol `N`.

```
N -> N 'and' N
N -> N 'or' N
N -> 'a' | 'b' | 'c'
```

An important concept for parsing is a *derivation*. For the input string *a and b or c* the following sequence of actions separated by the `=>` symbol represents a sequence of steps called a derivation:

```
N
=> N 'or' N
=> N 'or c'
=> N 'and' N 'or c'
=> N 'and b or c'
=> 'a and b or c'
```

In this derivation each line is called a *sentential form*. Furthermore, each line of the derivation applies a rule from the CFG in order to show that the input can, in fact, be derived from the start symbol N. In the above derivation, we restricted ourselves to only expand on the *rightmost* non-terminal in each sentential form. This method is called the *rightmost derivation* of the input using a CFG. An interesting property of a rightmost derivation is that if we arrange the derivation in reverse order:

```
'a and b or c'
=> N 'and b or c'       # use rule N -> a
=> N 'and' N 'or c'     # use rule N -> b
=> N 'or c'             # use rule N -> N and N
=> N 'or' N             # use rule N -> c
=> N                    # use rule N -> N or N
```

This derivation sequence exactly corresponds to the construction of the following parse tree from left to right one symbol at a time.

```
(N (N (N a)
      and
      (N b))
   or
   (N c))
```

However, a unique derivation sequence is not guaranteed. There can be many different derivations, and as we have seen before the number of derivations can be exponential in the input length. For example, there is another rightmost derivation that results in the following parse tree:

```
(N (N a)
   and
   (N (N b)
      or
      (N c)))

'a and b or c'
=> N 'and b or c'       # use rule N -> a
=> N 'and' N 'or c'     # use rule N -> b
=> N 'and' N 'or' N     # use rule N -> c
=> N 'and' N            # use rule N -> N or N
=> N                    # use rule N -> N and N
```

1.6.1 Shift Reduce Parsing

In order to build a parser, we need to create an algorithm that can perform the steps in the above rightmost derivation for any grammar and for any input string. Every CFG turns out to have an automata that is equivalent to it, called pushdown automata (just like regular expressions can be converted to finite-state automata). A pushdown automata is simply a finite-state automata with some additional memory in the form of a stack (or pushdown). This is a limited amount of memory since only the top of the stack is used by the machine. This provides an algorithm for parsing that is general for any given CFG and input string. The algorithm is called *shift-reduce* parsing which uses two data-structures: a buffer for input symbols and a stack for storing CFG symbols and is defined as follows:

1. Start with an empty stack and the buffer contains the input string.
2. Exit with success if the top of the stack contains the start symbol of the grammar and if the buffer is empty.
3. Choose between the following two steps (if the choice is ambiguous, choose one based on an oracle):
    - Shift a symbol from the buffer onto the stack.
    - If the top k symbols of the stack are $\alpha_1 \ldots \alpha_k$ which corresponds to the right-hand side of a CFG rule $A \rightarrow \alpha_1 \ldots \alpha_k$ then replace the top k symbols with the left-hand side non-terminal A.
4. Exit with failure if no action can be taken in previous step.
5. Else, go to Step 2.

For the CFG G shown earlier in this section and for the input *a and b or c* we show the individual steps in the shift reduce parsing algorithm:

Parse tree	Stack	Input	Action
		a and b or c	Init
a	a	and b or c	shift a
(N a)	N	and b or c	reduce `N -> a`
(N a) and	N and	b or c	shift and
(N a) and b	N and b	or c	shift b
(N a) and (N b)	N and N	or c	reduce `N -> b`
(N (N a) and (N b))	N	or c	reduce `N -> a`
(N (N a) and (N b)) or	N or	c	shift or
(N (N a) and (N b)) or c	N or c		shift c
(N (N a) and (N b)) or (N c)	N or N		reduce `N -> c`
(N (N (N a) and (N b)) or (N c))	N		reduce `N -> N or N`
(N (N (N a) and (N b)) or (N c))	N		Accept!

The same algorithm can also be applied for dependency parsing, as can be seen by the following example of using a shift reduce parser for dependency parsing. At each step the parser has a choice: either shift a new token into the stack, or combine the top two elements of the stack with a head → dependent link or a dependent ← head link. When using the shift-reduce algorithm in a statistical dependency parser it helps to combine a shift and reduce action when possible. Other variants that vary the link between parser actions and statistical decisions are discussed in (Nivre, 2008).

Dependency tree	Stack	Input	Action
root	root	a and b or c	Init
root a	root a	and b or c	shift a
root a and	root a and	b or c	shift and
root a and	root and	b or c	a ← and
root a and b	root and b	or c	shift b
root a and b	root and	or c	and → b
root a and b or	root and or	c	shift or
root a and b or	root or	c	and ← or
root a and b or c	root or c		shift c
root a and b or c	root or		or → c
root a and b or c	root		root → or

1.6.2 Hypergraphs and Chart Parsing

Shift-reduce parsing allows a linear time parse but requires access to an oracle. For general CFGs in the worse case such a parser might have to resort to backtracking, which means re-parsing the input which leads to a time that is exponential in the grammar size in the worst case. On the other hand, CFGs do have a worst case parsing algorithm that can run in $\mathcal{O}(n^3)$ where n is the length of the input. Variants of this algorithm are often used in statistical parsers that attempt to search the space of possible parse trees without the limitation of purely left to right parsing.

Our example CFG G:

```
N -> N 'and' N
N -> N 'or' N
N -> 'a' | 'b' | 'c'
```

is re-written into a new CFG G_c where the right hand side only contains up to two non-terminals. This is done by introducing two new non-terminals `N^` and `Nv`:

```
N -> N N^
N^ -> 'and' N
N -> N Nv
Nv -> 'or' N
N -> 'a' | 'b' | 'c'
```

A key insight into this family of parsing algorithms is that we can *specialize* the above CFG G_c to a particular input string by creating a new CFG which represents a compact encoding of all possible parse trees that are valid in grammar G_c for this particular input sentence. For example, for input string *a and b or c* this new CFG G_f that represents the *forest* of parse trees is shown below. Imagine that the input string is broken up into spans *0 a 1 and 2 b 3 or 4 c 5* so that *a* is span 0,1 and the string *b or c* is the span 2,5 in this string. The non-terminals in this forest grammar G_c include the span information. The different parse trees that can be generated using this grammar are the valid parse trees for the input sentence.

```
N[0,5] -> N[0,1] N^[1,5]
N[0,3] -> N[0,1] N^[1,3]
N^[1,3] -> 'and'[1,2] N[2,3]
N^[1,5] -> 'and'[1,2] N[2,5]
N[0,5] -> N[0,3] Nv[3,5]
N[2,5] -> N[2,3] Nv[3,5]
Nv[3,5] -> 'or'[3,4] N[4,5]
N[0,1] -> 'a'[0,1]
N[2,3] -> 'b'[2,3]
N[4,5] -> 'c'[4,5]
```

In this view, a parsing algorithm is defined as taking as input a CFG and an input string and producing a specialized CFG that is a compact representation of all legal parses for the input. A parser has to create all the valid specialized rules or alternatively create a path from the start symbol non-terminal that spans the entire string to the leaf nodes that are the input tokens.

Let us examine the steps the parser has to take to construct a specialized CFG. First let us consider the rules that generate only lexical items:

```
N[0,1] -> 'a'[0,1]
N[2,3] -> 'b'[2,3]
N[4,5] -> 'c'[4,5]
```

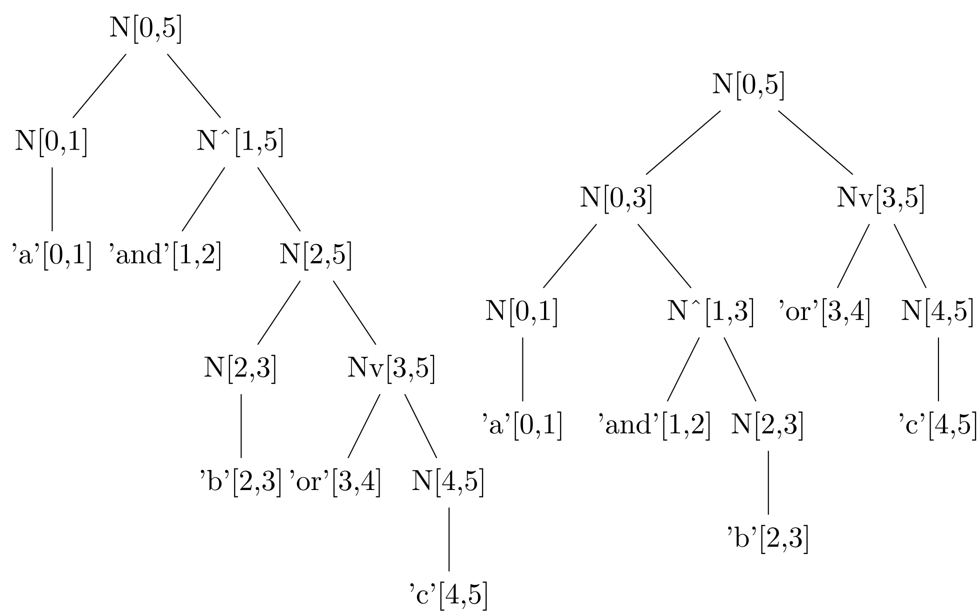


Figure 1.8. Parse trees embedded in the *specialized* CFG for a particular input string. The nodes with the same label, e.g. *N[0,5]* can be merged to form a hypergraph representation of all parses for the input.

These rules can be constructed by simply checking for the existence of rules of the type $N \rightarrow x$ for any input token x and creating a specialized rule for token x. In pseudo-code this step can be written as follows:

for $i = 0 \ldots n$ **do**
  if $N \rightarrow x$ with score s for any x spanning $i, i+1$ exists **then**
    add specialized rule $N[i, i+1] \rightarrow x[i, i+1]$ with score s
    written as $N[i, i+1] : s$
  end if
end for

The next step recursively creates new specialized rules based on previously created specialized rules. If we see that $Y[i, k]$ and $Z[k, j]$ exist as left-hand sides of previously created specialized rules then if there is a rule in the CFG of the type $X \rightarrow YZ$ we can infer that there should be a new specialized rule $X[i, j] \rightarrow Y[i, k]Z[k, j]$. Each non-terminal span is assigned a score s, $X[i, j] : s$. Only the highest scoring span for each non-terminal needs to be retained so $X[i, j] = \max_s X[i, j] : s$.

for $j = 2 \ldots n$ **do**
  for $i = j - 1 \ldots 0$ **do**
    for $k = i + 1 \ldots j$ **do**
      if $Y[i, k] : s_1$ and $Z[k, j] : s_2$ are in the specialized grammar **then**
        if $X \rightarrow YZ$ with score s exists in the original grammar **then**
          add specialized rule $X[i, j] \rightarrow Y[i, k]Z[k, j]$ with score $s + s_1 + s_2$
        end if
      end if
    end for
  end for
end for

This is called the CKY algorithm (named after Cocke, Kasami and Younger who all discovered it independently). It considers every span of every length, and splits up that span in every possible way to see if a CFG rule can be used to derive the span. Eventually we are guaranteed to find a rule that spans the entire input string, if such a rule exists. Examining the loop structure of the algorithm shows that it takes n^3 time for an input of size n. However, exhaustively listing all trees from the specialized CFG will take exponential time in the worst case (by the reasoning we already covered about the number of trees possible in the worst case for CFGs). However, picking the most likely tree by using supervised machine learning will take no more than n^3 time. The parser can be sped up even further by using A$*$ search rather than the exhaustive search used in the algorithm shown above. There is a rich choice of heuristics that can drive A$*$ search which can be used to provide faster observed times for parsing while the asymptotic worst case complexity remains the same as the CKY algorithm.

Starting from the start symbol that spans the entire string $S[0][n]$ with the highest score we can create the best scoring parse tree by expanding the right hand side of $S[0][n]$ and continue this process recursively to the terminal symbols. As we mentioned earlier in this process we always choose $X[i][j]$ rules with the highest score among all rules with $X[i][j]$ on the left-hand side.

For projective dependency parsing, the same algorithm can be used by creating a CFG that will produce dependency parses (as we showed in an earlier section). However, for dependency parsing, the above loop takes worst case n^5 time since each Y and Z is lexicalized and in the worst case there can be n different Y non-terminals and n different Z non-terminals giving us n^2 different possible combinations in the innermost loop of the above algorithm.

However for dependency parsing (Eisner, 1996) made an observation that for dependency parsing rather than using lexicalized non-terminals, another choice can be made to represent the different dependency trees for a span of input. The idea is to collect the left and the right dependents of each head independently, and combine them at a later stage. This leads to the notion of a "split-head" where the head word is split into two: one for the left and the other for the right dependents. In addition to the head word, in each item we store for each span, we store if the head is gathering left or right dependents, and if the item is complete (a complete item cannot be extended with more dependents). This provides a n^3 worst case dependency parsing algorithm. This also reduces the number of intermediate states considered by not allowing any interleaving of left and right dependencies unlike the CKY parser for dependency parsing.

The following pseudocode (from Ryan McDonald's thesis) describes the Eisner algorithm in full detail. The spans are stored in a chart data-structure C, e.g. $C[i][j]$ refers to the dependency analysis of span i, j. Incomplete spans are referred to as C^i, complete spans are C^c. Spans that are being grown towards left (adding left dependencies only) are referred to as $C_\leftarrow$, and similarly for spans growing to the right which are referred to as $C_\rightarrow$. For $C_\leftarrow[i][j]$ the head must be j and for $C_\rightarrow[i][j]$ the head must be i.

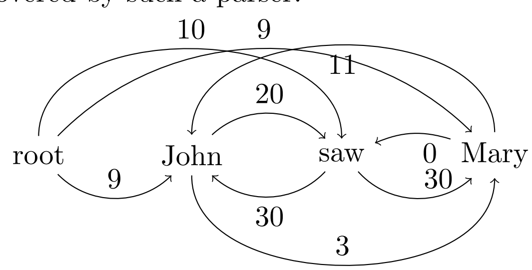
Initialize: for $s = 1..n$ chart $C^c_d[s][s] = 0.0$ for $d \in \{\leftarrow, \rightarrow\}$ and $c \in \{i, c\}$
for $k = 1 \ldots n$ **do**
  for $s = 1 \ldots n$ **do**
    $t = s + k$
    break if $t > n$
    first: create incomplete items
    $C^i_\leftarrow[s][t] = \max_{s \le r < t} C^c_\rightarrow[s][r] + C^c_\leftarrow[r+1][t] + s(t, s)$
    $C^i_\rightarrow[s][t] = \max_{s \le r < t} C^c_\rightarrow[s][r] + C^c_\leftarrow[r+1][t] + s(s, t)$
    second: create complete items
    $C^c_\leftarrow[s][t] = \max_{s \le r < t} C^c_\leftarrow[s][r] + C^i_\leftarrow[r][t]$
    $C^c_\rightarrow[s][t] = \max_{s \le r < t} C^i_\rightarrow[s][r] + C^c_\rightarrow[r][t]$
  end for
end for

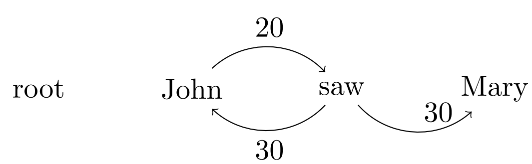
We assume a unique root node as the leftmost token (as before). The score of the best tree for the entire sentence is in $C^{c}_{\rightarrow}[1][n]$.

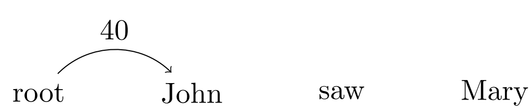
TODO: Show the running of the pseudo-code on the *a and b or c* example from previous section.

1.7 Minimum spanning trees and dependency parsing

Finding the optimum branching in a directed graph is closely related to the problem of finding a minimum spanning tree in an undirected graph. The directed graph case is of interest since it corresponds to a dependency tree which are always rooted and cannot have cycles. A pre-requisite is that each potential dependency link between words should have a score. In NLP, the tradition is to use the term "minimum spanning tree" to refer to the optimum branching problem in directed graphs. In the case of parsing with a dependency treebank we assume we have some model that can be used to provide such a score based on estimates of likelihood of each dependency link in the dependency tree. These scores can be used to find the minimum spanning tree which is the highest scoring dependency tree. The only problem is that the linear order of the words in the input is not taken into account in the MST formulation, and so crossing or non-projective dependencies can be recovered by such a parser.







1.8 Tagging and Chunking

This section covers tagging and chunking.

== dynamic programming for tagging

== application to noun-phrase chunking

1.9 Multilingual Issues: What is a token?

1.9.1 Tokenization, Case and Encoding

So far we have assumed that in a grammar or in a treebank the notion of a word, or more specifically that of a word token is well defined. However, this definition might be well-defined given a treebank or parser but variable across different parsers or treebanks. One example is the possessive marker and copula verb *'s* (a variant of *be*) in English. In English, a token is typically separated from other tokens by a space character. However, in a parser/treebank for English a word like *today's* or *There's* is treated as two independent tokens, *today* and *'s*, or *There* and *'s*. As the Penn Treebank annotation guidelines point out, the possessive marker can apply to some previous constituent and not just to the previous token:

```
(NP (NP (NP First)
        (PP of
            (NP America))
        's)
    operating results)
```

Similarly for the copula verb *'s*:

```
(S (NP-SBJ (EX There))
   (VP (VBZ 's)
       (NP-PRD (NP (NN nothing))
               (ADJP (RB very)
                     (JJ hot))))))
```

In some languages there are issues with upper and lower-case. It is tempting to lower-case all your treebank data and simply lower-case input texts for the parser. However, case can carry useful information. The token *Boeing* if it is previously unseen in the training data might look like a progressive verb like *singing* but the initial upper-case character makes it more likely to be a proper noun. However, depending on the type and amount of data available for training, some case information, such as selective lower-casing of sentence initial tokens, might have to be done to obtain reasonable estimates from the treebank. Low count tokens can be replaced with patterns that retain case information, e.g. if *Patagonia* appears only twice in the corpus then it can be replaced with *Xxx* to reflect that new unknown words that match the same pattern can be treated as known under this pattern. Similarly for cases like dates, times, IP addresses, URLs, etc.

For language scripts that are not encoded in ASCII the different encodings need to managed. In particular, the data the parser will be used on should be converted to the encoding that the treebank uses, or vice versa. There are often issues with the sentence terminator period *'.'* which in some text corpus might be an ASCII character but in another corpus it may be encoded in UTF-8. Some languages like

Chinese are encoded in different formats depending on the place where the text originates, e.g. GB, BIG5 or UTF-8 are all encodings you might find for Chinese text.

These are trivial issues, algorithmically speaking, compared to writing a parser, but in practice these issues can be quite challenging and time consuming, and while a full discussion of these issues is beyond the scope of this chapter, it does need to be pointed out that thinking about tokenization, case and encoding are pre-requisites for anyone wishing to write a parser or to get one working for a new language.

1.9.2 Word Segmentation

This section needs to be shortened a little.

The written form of many languages, including Chinese, do not have marks identifying words. Given the Chinese text "北京大学生比赛", a plausible segmentation would be "北京(Beijing)/大学生(university students)/比赛(competition)" (Competition among university students in Beijing). However, if "北京大学" is taken to mean Beijing University, the segmentation for the above character sequence might become "北京大学(Beijing University)/生(give birth to)/比赛(competition)" (Beijing University gives birth to competition), which is less plausible.

Word segmentation refers to the process of demarcating blocks in a character sequence such that the produced output is composed of separated tokens and is meaningful. Only if we have identified each word in a sentence, can part of speech tags (e.g. NNP or DT) then be assigned and the syntax tree for the whole sentence be built. In systems dealing with English or French, tokens are assumed to be already available since words have always been separated by spaces in these languages. While in Chinese, characters are written next to each other without marks identifying words.

Chinese word segmentation has a large community of researchers, and has resulted in three shared tasks: the SIGHAN bakeoffs [6], [7], [8].

There are many ways to approach the Chinese word segmentation problem, which can be classified into two main categories: the *dictionary-based matching* approach, the *sequence learning* approach.

The dictionary-based matching approach is simple and efficient. It uses a machine-readable lexical dictionary, which can be prepared beforehand, and it maps possible words in sentences to entries in the dictionary. The simplest and easiest-to-implement algorithm is the greedy longest matching method, traversing from left to right in the current test sentence, greedily taking the longest match based on the words in the lexical dictionary. It has been officially used as the model to produce baseline scores in previous SIGHAN bakeoffs [6], [7], [8]. One major drawback of the dictionary-based matching approach is that, in order to get a high-quality result, the dictionary has to be as complete as possible. Also, dictionary matching using the greedy longest match can ignore plausible word sequences in favor of implausible ones. If we assume that the dictionary is collected from a training data set containing sentences with human-annotated word sequences, then a simple extension of the dictionary approach is to find the most likely sequence of words, based on the probability of words obtained from the training data. Each character sequence is divided into all possible candidate $w_1, \ldots, w_n$ word sequences based on the dictionary collected from the training data and then the likelihood of each candidate word sequence is:

$$P(w_1, \ldots, w_N) = \prod_i P(w_i)$$

To find the most likely word sequence efficiently a dynamic programming algorithm is used, which computes the likelihood of word sequences from left to right, one character at a time.

In the sequence learning approach, each character is assigned a particular tag (similar to a part of speech tag but specialized for word segmentation), indicating the position of that character within a word (e.g. either inside, outside or at the beginning of a word). Patterns or statistical information are obtained from the tagged training examples with machine learning techniques, and this information is then used to predict tags for unseen test data so that the optimal tag sequence is produced. Various tagsets have been explored for Chinese word segmentation [9]. The two most typical types of tagset are the 3-tag "IOB" set and the 4-tag "BMES" set.

In the "IOB" tagset, the first character of a multi-character word is assigned the "B" (**B**eginning) tag, and each remaining character of the multi-character word is assigned the "I" (**I**nside) tag. For a single-character word, that character is assigned the "O" (**O**utside) tag, indicating that character is outside a multi-character word. While in the "BMES" tagset, for each multi-character word, its first character is given the "B" (**B**eginning) tag , its last character is given the "E" (**E**nd) tag, while each remaining character is given the "M" (**M**iddle) tag. In addition, for a single-character word, "S" (**S**ingle) is used as its tag (same as the "O" tag in "IOB" tagset). For instance, the sentence "新华社/上海/二月/十日/电" (Report from Xinhua News Agency on February 10th in Shanghai) is tagged as follows:

- With "IOB" Tagset: 新-B 华-I 社-I 上-B 海-I 二-B 月-I 十-B 日-I 电-O
- With "BMES" Tagset: 新-B 华-M 社-E 上-B 海-E 二-B 月-E 十-B 日-E 电-S

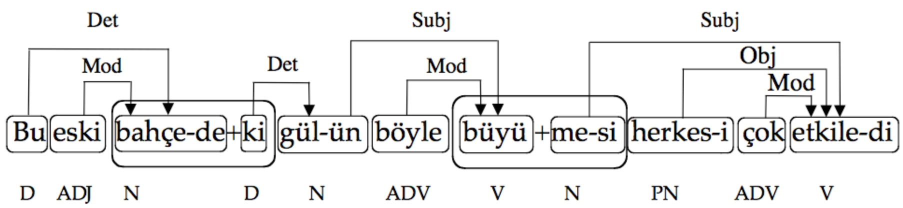
Given the assignment of tags to characters, word segmentation can be treated exactly like the part of speech tagging task, and the same algorithms for training and prediction can be used to segment sentences into word sequences.

One interesting approach to Chinese parsing has been to parse the character sequence directly. The parser itself assigns word boundaries as part of the parsing process, where certain non-terminals in the tree that span a group of characters implicitly specify the word boundaries. (citation)

1.9.3 Morphology

In many languages the notion of splitting up tokens using spaces is problematic since each word can contain several components, called morphemes, such that the meaning of a word can be thought of as composed of the combination of the meanings of the morphemes. A word must now be thought of as being decomposed into a stem combined with several morphemes.

For example, the following dependency analysis from the Turkish treebank shows that the syntactic dependencies need to be aware of the morphemes within words. In this example, morpheme boundaries within a word are shown using the + symbol. Morphemes, and not words, are used as heads and dependents.



Turkish, Finnish and other agglutinative languages have this property of entire clauses being combined together with morphemes to create very complex words.

Inflectional languages like Czech or Russian are not as extreme but also suffer from the problem that many different morphemes are used to mark grammatical case, gender, etc. and each type of morpheme can be orthogonal to the others (so they can independently co-occur). For instance, as per (Hajic and Hladka, 2000) most adjectives in Czech can potentially form all 4 genders, all 7 case markers, all 3 degrees of comparison, and can be either positive or negative in polarity. This results in 336 possible inflected word forms just for adjectives. In addition to large number of word forms, each inflected word can be ambiguously segmented into morphemes with different analyses. Quite apart from syntactic ambiguity the parser now also has to deal with morphological ambiguity.

In order to tackle the disambiguation problem for morphology, the problem of splitting a word into the most likely sequence of morphemes can be reduced to a (very complex) part of speech tagging task. In this case each word is to be tagged with a complex part of speech that encodes the various morphemes. For instance a part of speech of `V--M-3----` can indicate that each word can have morphemes that inflect the word along 10 different dimensions, and in this case the stem is a verb (V) with masculine gender (M) and in third person (3) and the other types of morphemes are assigned `-` which indicates that they do not occur in this analysis. The part of speech tagger has to produce this complex tag – and this is typically done by training separate sub-classifiers for each component of the part of speech tag, and combining the output for tagging each word. The word itself is not split into morphemes, but each word is tagged with a part of speech tag that encodes a lot of information about the morphemes. This enriched tag set can be a rich source of features for a statistical parser for a highly inflected language.

In Turkish, Finnish, and other agglutinative languages the morphemes must be separated by spaces since the morphemes can possibly participate in independent ways in the dependency structure. So in such languages a morphological analyzer is typically combined with statistical classifiers to predict the most likely stem plus morpheme breakdown for a given word. Recently there has been a lot of interest in unsupervised learning for morphological analysis and tagging for agglutinative languages like Finnish (cite Morfessor).

Once we have a morphological analysis of words in the input sentence, how best to use this information to generalize beyond ordinary tokens for statistical parsing?

The smoothing using the morphological tagger is handled as follows. The statistical parser has various probability models associated with it. One of the most crucial models is the one that decides on parser actions by using statistics collected on pairs of words. For example, consider probability P to be the probability of combination of two words w and w' with label l. Here w and w' are the inflected forms of the word, while p and p' are selected elements of the disambiguated morphological analysis of the inflected forms taken from the output of the morphological tagger described above. Based on the part of speech, we might want to select different components of the morphological analysis. For example, for nouns we might want to pick the final element which usually corresponds to the case marking, while for verbs we might want to pick the stem which is the first element of the analysis. We have the flexibility in the parser to choose which element should be picked. The best results in (Sarkar and Han, 2002) were obtained when the stem was chosen rather than any of the suffix or prefix morphemes.

$$Pr(l, p', w' \mid w, p)$$

We decompose this conditional probability into the following components:

$$\begin{aligned} &Pr(l, p', w' \mid w, p) = \\ &\quad Pr(l \mid w, p) \times \\ &\quad Pr(p' \mid l, w, p) \times \\ &\quad Pr(w' \mid p', l, w, p) \end{aligned}$$

For each of the equations above, we use a backoff model which is used to handle sparse data problems. We compute a backoff model as follows. Let e_1 stand for the original lexicalized model and e_2 be the backoff level which only uses the output of the morphological tagger described above:

$$\begin{aligned} Pr_{e_1} &= Pr(l \mid \ldots, w, p) \\ Pr_{e_2} &= Pr(l \mid \ldots, p) \end{aligned}$$

The backoff model is computed as follows:

$$\lambda(c) \times Pr_{e_1} + (1 - \lambda(c)) \times Pr_{e_2}$$

where $\lambda(c) = \frac{c}{(c+D)}$. $c = \mathit{count}(r)$ is the total count for each conditional probability model P, where $P(l \mid r)$. D is the *diversity* of Pr_{e_1}. *diversity* is defined as the number of distinct counts for Pr_{e_1}. Note that this backoff model is implemented for each lexicalized model used in the statistical parser.

Also discuss (Cowan and Collins, EMNLP 2005) Morphology and Reranking for the Statistical Parsing of Spanish.

1.10 Advanced Topics

An eclectic selection of advanced topics in statistical parsing.

== partition function for dependency parser training: matrix-tree theorem

== forest re-ranking

== combining syntax parsing and semantic parsing

== selecting an optimal set of feature functions for each language

1.11 Summary

In this chapter we have learned about the following ...

Acknowledgments

Thanks to the School of Informatics at the University of Edinburgh for hosting me during my sabbatical year away from Simon Fraser University. The majority of this chapter was written in Edinburgh.

BIBLIOGRAPHY

[1] M. Collins, "Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms," in *Proceedings of the Empirical Methods in Natural Language Processing (EMNLP)*, (Philadelphia, PA, USA), pp. 1–8, ACL, July 2002.

[2] J. Lafferty, A. McCallum, and F. Pereira, "Conditional random fields: Probabilistic models for segmenting and labeling sequence data," in *Proceedings of the 18th International Conf. on Machine Learning (ICML)*, pp. 282–289, 2001.

[3] F. Rosenblatt, "The perception: a probabilistic model for information storage and organization in the brain," *Psychological Review*, vol. 65, no. 6, pp. 386–408, 1958.

[4] Y. Freund and R. E. Schapire, "Large margin classification using the perceptron algorithm," *Mach. Learn.*, vol. 37, no. 3, pp. 277–296, 1999.

[5] M. Collins, "Ranking algorithms for named entity extraction: Boosting and the voted perceptron," in *Proceedings of ACL 2002*, (Philadelphia, PA, USA), pp. 489–496, ACL, July 2002.

[6] R. Sproat and T. Emerson, "The 1st international chinese word segmentation bakeoff," in *Proceedings of the 2nd SIGHAN Workshop on Chinese Language Processing*, (Sapporo, Japan), pp. 123–133, ACL, July 2003.

[7] T. Emerson, "The 2nd international chinese word segmentation bakeoff," in *Proceedings of the 4th SIGHAN Workshop on Chinese Language Processing*, (Jeju Island, Korea), pp. 123–133, October 2005.

[8] G.-A. Levow, "The 3rd international chinese language processing bakeoff," in *Proceedings of the 5th SIGHAN Workshop on Chinese Language Processing*, (Sydney, Australia), pp. 108–117, ACL, July 2006.

[9] N. Xue, "Chinese word segmentation as character tagging," *Computational Linguistics and Chinese Language Processing*, vol. 8, no. 1, pp. 29–48, 2003.

[10] I. Bhattacharya, L. Getoor, and Y. Bengio, "Unsupervised sense disambiguation using bilingual probabilistic models," in *Proceedings of the 42nd Meeting of the Association for Computational Linguistics (ACL'04), Main Volume*, (Barcelona, Spain), pp. 287–294, July 2004.

[11] R. Caruana and A. Niculescu-Mizil, "An empirical comparison of supervised learning algorithms using different performance metrics," Tech. Rep. TR2005-1973, Cornell University, 2005.

[12] K.-J. Chen and S.-H. Liu, "Word identification for mandarin chinese sentences," in *Proceedings of the 14th conference on Computational linguistics*, (Morristown, NJ, USA), pp. 101–107, ACL, 1992.

[13] W. Chen, Y. Zhang, and H. Isahara, "Chinese named entity recognition with conditional random fields," in *Proceedings of the 5th SIGHAN Workshop on Chinese Language Processing*, (Sydney, Australia), pp. 118–121, ACL, July 2006.

[14] M. Collins, "Parameter estimation for statistical parsing models: theory and practice of distribution-free methods," in *New developments in parsing technology*, pp. 19–55, Norwell, MA, USA: Kluwer Academic Publishers, 2004.

[15] M. Collins, "Discriminative reranking for natural language parsing," in *Proceedings of the 17th International Conference on Machine Learning (ICML)*, pp. 175–182, 2000.

[16] M. Collins and B. Roark, "Incremental parsing with the perceptron algorithm," in *Proceedings of the 42nd Meeting of the Association for Computational Linguistics (ACL'04), Main Volume*, (Barcelona, Spain), pp. 111–118, July 2004.

[17] A. Culotta, A. McCallum, and J. Betz, "Integrating probabilistic extraction models and data mining to discover relations and patterns in text," in *Proceedings of the Human Language Technology Conference of the NAACL, Main Conference*, (New York City, USA), pp. 296–303, ACL, June 2006.

[18] L. Gillick and S. Cox, "Some statistical issues in the comparison of speech recognition algorithms," in *Proc. of IEEE Conf. on Acoustics, Speech and Sig. Proc.*, (Glasgow, Scotland), pp. 532–535, May 1989.

[19] A. Globerson, T. Y. Koo, X. Carreras, and M. Collins, "Exponentiated gradient algorithms for log-linear structured prediction," in *ICML '07: Proceedings of the 24th international conference on Machine learning*, (New York, NY, USA), pp. 305–312, ACM, 2007.

[20] S. I. Hill and R. C. Williamson, "Convergence of exponentiated gradient algorithms," in *IEEE Trans. Signal Processing*, vol. 49, pp. 1208–1215, June 2001.

[21] J. Hockenmaier and M. Steedman, "Generative models for statistical parsing with combinatory categorial grammar," in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL)*, (Philadelphia, USA), pp. 335–342, ACL, July 2002.

[22] Z. Huang, M. Harper, and W. Wang, "Mandarin part-of-speech tagging and discriminative reranking," in *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pp. 1093–1102.

[23] J. Kazama and K. Torisawa, "A new perceptron algorithm for sequence labeling with non-local features," in *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pp. 315–324.

[24] B. Kégl and G. Lapalme, eds., *Advances in Artificial Intelligence, 18th Conference of the Canadian Society for Computational Studies of Intelligence, Canadian AI 2005, Victoria, Canada, May 9-11, 2005, Proceedings*, vol. 3501 of *Lecture Notes in Computer Science*, Springer, 2005.

[25] K. Kim, Y. Song, and G. G. Lee, "POSBIOTM/W: A development workbench for machine learning oriented biomedical text mining system," in *Proceedings of HLT/EMNLP 2005 Interactive Demonstrations*, (Vancouver, British Columbia, Canada), pp. 36–37, ACL, October 2005.

[26] J. Kivinen and M. Warmuth, "Exponentiated gradient versus gradient descent for linear predictors," Tech. Rep. UCSC-CRL-94-16, UC Santa Cruz, 1994.

[27] T. Kudo, Y. Kaoru, and M. Yuji, "Applying conditional random fields to japanese morphological analysis," in *Proceedings of EMNLP 2004* (D. Lin and D. Wu, eds.), (Barcelona, Spain), pp. 230–237, ACL, July 2004.

[28] P. Liang, "Semi-supervised learning for natural language," Master's thesis, Massachusetts Institute of Technology, 2005.

[29] Y. Liu, Q. Tan, and X. Shen, "Chinese national standard gb13715: Contemporary chinese word segmentation standard used for information processing," in *Information Processing with Contemporary Mandarin Chinese Word Segmentation Standard and Automatic Segmentation Method*, Tsinghua University Press, 1994.

[30] A. Novikoff, "On convergence proofs on perceptrons," in *Proceedings of the Symposium on Mathematical Theory of Automata*, pp. 615–622, Polytechnic Institute of Brooklyn, 1962.

[31] C. P. Papageorgiou, "Japanese word segmentation by hidden markov model," in *HLT '94: Proceedings of the workshop on Human Language Technology*, (Morristown, NJ, USA), pp. 283–288, ACL, 1994.

[32] F. Peng, F. Feng, and A. McCallum, "Chinese segmentation and new word detection using conditional random fields," in *Proceedings of Coling 2004*, (Geneva, Switzerland), pp. 562–568, August 2004.

[33] S. D. Pietra, V. D. Pietra, and J. Lafferty, "Inducing features of random fields," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 19, no. 4, pp. 380–393, 1997.

[34] A. Ratnaparkhi, "A maximum entropy model for part-of-speech tagging," in *Proceedings of the Conference on Empirical Methods in Natural Language Processing* (E. Brill and K. Church, eds.), (Somerset, New Jersey), pp. 133–142, Association for Computational Linguistics, 1996.

[35] L. R. Rabiner, "A tutorial on hidden markov models and selected applications in speech recognition," *Proceedings of the IEEE*, vol. 77, pp. 257–286, Feburary 1989.

[36] E. F. T. K. Sang and J. Veenstra, "Representing text chunks," in *Proceedings of EACL 1999*, (Bergen, Norway), pp. 173–179, ACL, June 1999.

[37] F. Sha and F. Pereira, "Shallow parsing with conditional random fields," in *NAACL '03: Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology*, (Morristown, NJ, USA), pp. 134–141, ACL, 2003.

[38] H. Shen and A. Sarkar, "Voting between multiple data representations for text chunking," in *Canadian Conference on AI*, pp. 389–400, 2005.

[39] L. Shen, A. Sarkar, and F. J. Och, "Discriminative reranking for machine translation," in *HLT-NAACL 2004: Main Proceedings*, (Boston, Massachusetts, USA), pp. 177–184, ACL, May 2 - May 7 2004.

[40] R. Sproat, W. Gale, C. Shih, and N. Chang, "A stochastic finite-state word-segmentation algorithm for chinese," *Comput. Linguist.*, vol. 22, no. 3, pp. 377–404, 1996.

[41] A. Stolcke, "SRILM – an extensible language modeling toolkit," in *Proceedings of the ICSLP*, vol. 2, (Denver, Colorado), pp. 901–904, 2002.

[42] M. Sun and B. K. T'sou, "Ambiguity resolution in chinese word segmentation," in *Proceedings of the 10th Pacific Asia Conference on Language, Information and Computation*, (Hong Kong, China), pp. 121–126, City University of Hong Kong, December 1995.

[43] C. Sutton and A. Mccallum, "An introduction to conditional random fields for relational learning," in *Introduction to Statistical Relational Learning*, MIT Press, 2006.

[44] C. Thompson, S. Patwardhan, and C. Arnold, "Generative models for semantic role labeling," in *Senseval-3: 3rd International Workshop on the Evaluation of Systems for the Semantic Analysis of Text*, (Barcelona, Spain), pp. 235–238, ACL, July 2004.

[45] A. J. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," in *IEEE Transactions on Information Theory*, vol. 13, pp. 260–269, April 1967.

[46] M. Wang and Y. Shi, "Using part-of-speech reranking to improve chinese word segmentation," in *Proceedings of the 5th SIGHAN Workshop on Chinese Language Processing*, (Sydney, Australia), pp. 205–208, ACL, July 2006.

[47] A. Wu, "Customizable segmentation of morphologically derived words in chinese," *Computational Linguistics and Chinese Language Processing*, vol. 8, no. 1, pp. 1–27, 2003.

[48] S. Y. T.-S. Chua and L. Jimin, "An agent-based approach to chinese named entity recognition," in *Proceedings of the 19th international conference on Computational linguistics*, (Morristown, NJ, USA), pp. 1–7, ACL, 2002.

[49] S. Yu, H. Duan, B. Swen, and B.-B. Chang, "Specification for corpus processing at peking university: Word segmentation, pos tagging and phonetic notation," *Journal of Chinese Language and Computing*, vol. 13, 2003.

[50] X. Yu, M. Carpuat, and D. Wu, "Boosting for chinese named entity recognition," in *Proceedings of the 5th SIGHAN Workshop on Chinese Language Processing*, (Sydney, Australia), pp. 150–153, ACL, July 2006.

[51] H. Zhang, T. Liu, J. Ma, and X. Liu, "Chinese word segmentation with multiple postprocessors in HIT-IRLab," in *Proceedings of the 4th SIGHAN Workshop on Chinese Language Processing*, (Jeju Island, Korea), pp. 172–175, Asian Federation of Natural Language Processing, Octber 2005.

[52] R. Zhang, G. Kikui, and E. Sumita, "Subword-based tagging by conditional random fields for chinese word segmentation," in *Proceedings of the Human Language Technology Conference of the NAACL, Companion Volume: Short Papers*, (New York City, USA), pp. 193–196, ACL, June 2006.

[53] Y. Zhang and S. Clark, "Chinese segmentation with a word-based perceptron algorithm," in *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*, (Prague, Czech Republic), pp. 840–847, ACL, June 2007.

[54] H. Zhao, C.-N. Huang, and M. Li, "An improved chinese word segmentation system with conditional random field," in *Proceedings of the 5th SIGHAN Workshop on Chinese Language Processing*, (Sydney, Australia), pp. 162–165, ACL, July 2006.

[55] J. Zhou, L. He, X. Dai, and J. Chen, "Chinese named entity recognition with a multi-phase model," in *Proceedings of the 5th SIGHAN Workshop on Chinese Language Processing*, (Sydney, Australia), pp. 213–216, ACL, July 2006.

[56] D. Song and A. Sarkar, "Training a perceptron with global and local features for chinese word segmentation," in *Proceedings of the 6th SIGHAN Workshop on Chinese Language Processing*, pp. 143–146, 2008.

[57] A. Globerson, T. Koo, X. Carreras, and M. Collins, "Exponentiated gradient algorithms for log-linear structured prediction," in *ICML*, pp. 305–312, 2007.

[58] D. Song, "Experimental comparison of discriminative learning approaches for chinese word segmentation," Master's thesis, Simon Fraser University, 2008.