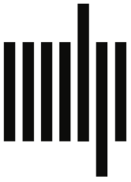
Foundations of Statistical Natural Language Processing

Christopher Manning and Hinrich Schuetze



The MIT Press

From The MIT Press

MITCogNet

Second printing with corrections 1999
Third printing 2000, fourth printing 2001
Fifth printing with corrections

Typeset in 10/13 Lucida Bright by the authors using LaTeX 2ε.
Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Information

Manning, Christopher D.
Foundations of statistical natural language processing / Christopher D. Manning, Hinrich Schütze.
p. cm.
Includes bibliographical references (p.) and index.
ISBN 0-262-13360-1
1. Computational linguistics—Statistical methods. I. Schütze, Hinrich. II. Title.
P98.5.S83M36 1999
410'.285—dc21 99-21137
CIP

10 9 8 7 6

12 *Probabilistic Parsing*

CHUNKING

THE PRACTICE of parsing can be considered as a straightforward implementation of the idea of *chunking* – recognizing higher level units of structure that allow us to compress our description of a sentence. One way to capture the regularity of chunks over different sentences is to learn a grammar that explains the structure of the chunks one finds. This

GRAMMAR INDUCTION

is the problem of *grammar induction*. There has been considerable work on grammar induction, because it is exploring the empiricist question of how to learn structure from unannotated textual input, but we will not cover it here. Suffice it to say that grammar induction techniques are reasonably well understood for finite state languages, but that induction is very difficult for context-free or more complex languages of the scale needed to handle a decent proportion of the complexity of human languages. It is not hard to induce *some* form of structure over a corpus of text. Any algorithm for making chunks – such as recognizing common subsequences – will produce some form of chunked representation of sentences, which we might interpret as a phrase structure tree. However, most often the representations one finds bear little resemblance to the kind of phrase structure that is normally proposed in linguistics and NLP.

Now, there is enough argument and disagreement within the field of syntax that one might find *someone* who has proposed syntactic structures similar to the ones that the grammar induction procedure which you have sweated over happens to produce. This can and has been taken as evidence for that model of syntactic structure. However, such an approach has more than a whiff of circularity to it. The structures found depend on the implicit inductive bias of the learning program. This suggests another tack. We need to get straight what structure we expect our

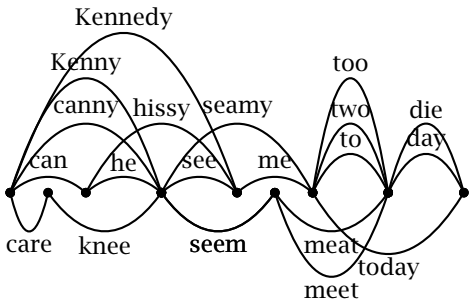


Figure 12.1 A word lattice (simplified).

model to find *before* we start building it. This suggests that we should begin by deciding what we want to do with parsed sentences. There are various possible goals: using syntactic structure as a first step towards semantic interpretation, detecting phrasal chunks for indexing in an IR system, or trying to build a probabilistic parser that outperforms *n*-gram models as a language model. For any of these tasks, the overall goal is to produce a system that can place a provably useful structure over arbitrary sentences, that is, to build a *parser*. For this goal, there is no need to insist that one begins with a *tabula rasa*. If one just wants to do a good job at producing useful syntactic structure, one should use all the prior information that one has. This is the approach that will be adopted in this chapter.

PARSER

The rest of this chapter is divided into two parts. The first introduces some general concepts, ideas, and approaches of broad general relevance, which turn up in various places in the statistical parsing literature (and a couple which should turn up more often than they do). The second then looks at some actual parsing systems that exploit some of these ideas, and at how they perform in practice.

12.1 Some Concepts

12.1.1 Parsing for disambiguation

There are at least three distinct ways in which one can use probabilities in a parser:

WORD LATTICE

- **Probabilities for determining the sentence.** One possibility is to use a parser as a language model over a *word lattice* in order to determine what sequence of words running along a path through the lattice has highest probability. In applications such as speech recognizers, the actual input sentence is uncertain, and there are various hypotheses, which are normally represented by a word lattice as in figure 12.1.[1] The job of the parser here is to be a language model that tries to determine what someone probably said. A recent example of using a parser in this way is (Chelba and Jelinek 1998).

- **Probabilities for speedier parsing.** A second goal is to use probabilities to order or prune the search space of a parser. The task here is to enable the parser to find the best parse more quickly while not harming the quality of the results being produced. A recent study of effective methods for achieving this goal is (Caraballo and Charniak 1998).

- **Probabilities for choosing between parses.** The parser can be used to choose from among the many parses of the input sentence which ones are most likely.

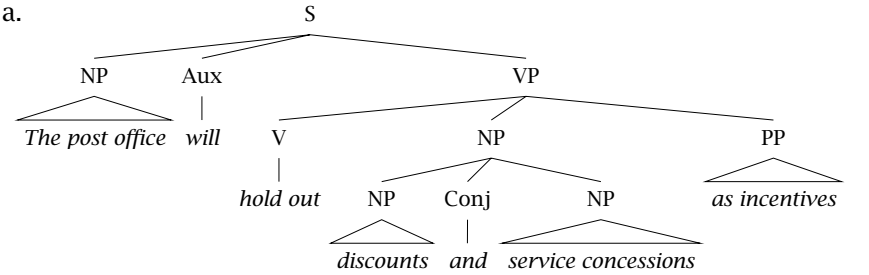
In this section, and in this chapter, we will concentrate on the third use of probabilities over parse trees: using a statistical parser for disambiguation.

Capturing the tree structure of a particular sentence has been seen as key to the goal of disambiguation – the problem we discussed in chapter 1. For instance, to determine the meaning of the sentence in (12.1), we need to determine what are the meaningful units and how they relate. In particular we need to resolve ambiguities such as the ones represented in whether the correct parse for the sentence is (12.2a) or (12.2b), (12.2c) or (12.2d), or even (12.2e).

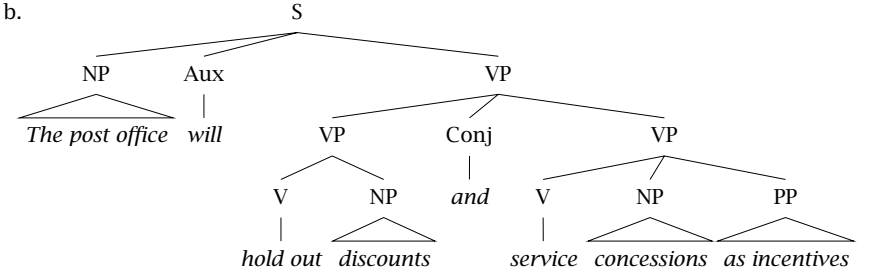
(12.1) The post office will hold out discounts and service concessions as incentives.

1. Alternatively, they may be represented by an *n-best list*, but that has the unfortunate effect of multiplying out ambiguities in what are often disjoint areas of uncertainty in the signal.

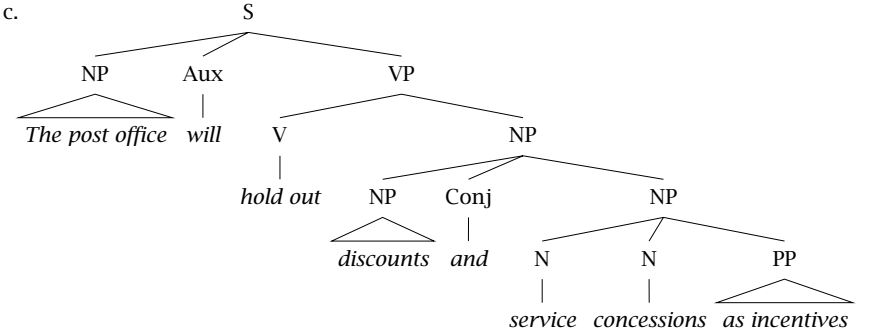
(12.2) a.


S
NP
Aux
VP
The post office
will
V
NP
PP
hold out
NP
Conj
NP
as incentives
discounts
and
service concessions

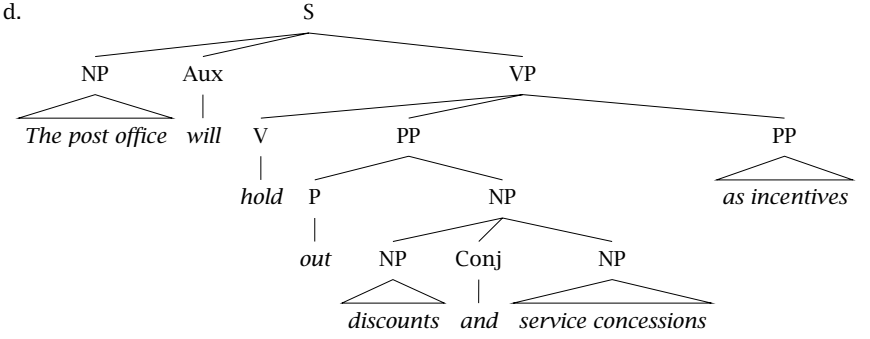
b.


S
NP
Aux
VP
The post office
will
VP
Conj
VP
V
NP
and
V
NP
PP
hold out
discounts
service
concessions
as incentives

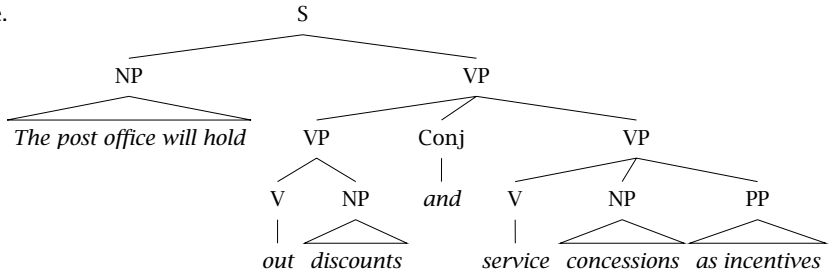
c.


S
NP
Aux
VP
The post office
will
V
NP
hold out
NP
Conj
NP
discounts
and
N
N
PP
service
concessions
as incentives

d.


S
NP
Aux
VP
The post office
will
V
PP
PP
hold
P
NP
as incentives
out
NP
Conj
NP
discounts
and
service concessions

e.



One might get the impression from computational linguistics books that such ambiguities are rare and artificial, because most books contain the same somewhat unnatural-sounding examples (ones about pens and boxes, or seeing men with telescopes). But that's just because simple short examples are practical to use. Such ambiguities are actually ubiquitous. To provide some freshness in our example (12.1), we adopted the following approach: we randomly chose a *Wall Street Journal* article, and used the first sentence as the basis for making our point. Finding ambiguities was not difficult.[2] If you are still not convinced about the severity of the disambiguation problem, then you should immediately do exercise 12.1 before continuing to read this chapter.

What is one to do about all these ambiguities? In classical categorical approaches, some ambiguities are seen as genuine syntactic ambiguities, and it is the job of the parser to return structures corresponding to all of these, but other weird things that one's parser spits out are seen as faults of the grammar, and the grammar writer will attempt to refine the grammar, in order to generate less crazy parses. For instance, the grammar writer might feel that (12.2d) should be ruled out, because *hold* needs an object noun phrase, and enforce that by a subcategorization frame placed on the verb *hold*. But actually that would be a mistake, because then the parser would not be able to handle a sentence such as: *The flood waters reached a height of 8 metres, but the sandbags held.*

In contrast, a statistically-minded linguist will not be much interested in how many parses his parser produces for a sentence. Normally there is still some categorical base to the grammar and so there is a fixed finite

2. We refrained from actually *using* the first sentence, since like so many sentences in newspapers, it was rather long. It would have been difficult to fit trees for a 38 word sentence on the page. But for reference, here it is: *Postmaster General Anthony Frank, in a speech to a mailers' convention today, is expected to set a goal of having computer-readable bar codes on all business mail by 1995, holding out discounts and service concessions as incentives.*

number of parses, but statistically-minded linguists can afford to be quite licentious about what they allow into their grammar, and so they usually are. What is important is the probability distribution over the parses generated by the grammar. We want to be able to separate out the few parses that are likely to be correct from the many that are syntactically possible, but extremely unlikely. In many cases, we are just interested in "the best parse," which is the one deemed to be most likely to be correct. Statistical parsers generally disambiguate and rate how likely different parses are as they parse, whereas in conventional parsers, the output trees would normally be sent to downstream models of semantics and world knowledge that would choose between the parses. A statistical parser usually disambiguates as it goes by using various extended notions of word and category collocation as a surrogate for semantic and world knowledge. This implements the idea that the ways in which a word tends to be used gives us at least some handle on its meaning.

12.1.2 Treebanks

We mentioned earlier that pure grammar induction approaches tend not to produce the parse trees that people want. A fairly obvious approach to this problem is to give a learning tool some examples of the kinds of parse trees that are wanted. A collection of such example parses is referred to as a *treebank*. Because of the usefulness of collections of correctly-parsed sentences for building statistical parsers, a number of people and groups have produced treebanks, but by far the most widely used one, reflecting both its size and readily available status, is the *Penn Treebank*.

TREEBANK

PENN TREEBANK

An example of a Penn Treebank tree is shown in figure 12.2. This example illustrates most of the major features of trees in the Penn treebank. Trees are represented in a straightforward (Lisp) notation via bracketing. The grouping of words into phrases is fairly flat (for example there is no disambiguation of compound nouns in phrases such as *Arizona real estate loans*), but the major types of phrases recognized in contemporary syntax are fairly faithfully represented. The treebank also makes some attempt to indicate grammatical and semantic functions (the **-SBJ** and **-LOC** tags in the figure, which are used to tag the subject and a locative, respectively), and makes use of empty nodes to indicate understood subjects and extraction gaps, as in the understood subject of the adverbial clause in the example, where the empty node is marked as *. In table 12.1,

```
( (S (NP-SBJ The move)
     (VP followed
         (NP (NP a round)
             (PP of
                 (NP (NP similar increases)
                     (PP by
                         (NP other lenders))
                     (PP against
                         (NP Arizona real estate loans)))))
         ,
         (S-ADV (NP-SBJ *)
                (VP reflecting
                    (NP (NP a continuing decline)
                        (PP-LOC in
                                (NP that market))))))
     .))
```

Figure 12.2 A Penn Treebank tree.

S	Simple clause (sentence)	CONJP	Multiword conjunction phrases
SBAR	S′ clause with complementizer	FRAG	Fragment
SBARQ	*Wh*-question S′ clause	INTJ	Interjection
SQ	Inverted *Yes/No* question S′ clause	LST	List marker
SINV	Declarative inverted S′ clause	NAC	Not A Constituent grouping
ADJP	Adjective Phrase	NX	Nominal constituent inside NP
ADVP	Adverbial Phrase	PRN	Parenthetical
NP	Noun Phrase	PRT	Particle
PP	Prepositional Phrase	RRC	Reduced Relative Clause
QP	Quantifier Phrase (inside NP)	UCP	Unlike Coordinated Phrase
VP	Verb Phrase	X	Unknown or uncertain
WHNP	*Wh*- Noun Phrase	WHADJP	*Wh*- Adjective Phrase
WHPP	*Wh*- Prepositional Phrase	WHADVP	*Wh*- Adverb Phrase

Table 12.1 Abbreviations for phrasal categories in the Penn Treebank. The common categories are gathered in the left column. The categorization includes a number of rare categories for various oddities.

we summarize the phrasal categories used in the Penn Treebank (which basically follow the categories discussed in chapter 3).

One oddity, to which we shall return, is that complex noun phrases are represented by an NP-over-NP structure. An example in figure 12.2 is the NP starting with *similar increases*. The lower NP node, often referred to as the 'baseNP' contain just the head noun and preceding material such as determiners and adjectives, and then a higher NP node (or sometimes two) contains the lower NP node and following arguments and modifiers. This structure is wrong by the standards of most contemporary syntactic theories which argue that NP postmodifiers belong with the head under some sort of N′ node, and lower than the determiner (section 3.2.3). On the other hand, this organization captures rather well the notion of *chunks* proposed by Abney (1991), where, impressionistically, the head noun and prehead modifiers seem to form one chunk, whereas phrasal postmodifiers are separate chunks. At any rate, some work on parsing has directly adopted this Penn Treebank structure and treats baseNPs as a unit in parsing.

CHUNKING

Even when using a treebank, there is still an induction problem of extracting the grammatical knowledge that is implicit in the example parses. But for many methods, this induction is trivial. For example, to determine a PCFG from a treebank, we need do nothing more than count the frequencies of local trees, and then normalize these to give probabilities.

Many people have argued that it is better to have linguists constructing treebanks than grammars, because it is easier to work out the correct parse of individual actual sentences than to try to determine (often largely by intuition) what all possible manifestations of a certain rule or grammatical construct are. This is probably true in the sense that a linguist is unlikely to immediately think of all the possibilities for a construction off the top of his head, but at least an implicit grammar must be assumed in order to be able to treebank. In multiperson treebanking projects, there has normally been a need to make this grammar explicit. The treebanking manual for the Penn Treebank runs to over 300 pages.

12.1.3 Parsing models vs. language models

The idea of parsing is to be able to take a sentence s and to work out parse trees for it according to some grammar G. In probabilistic parsing, we would like to place a ranking on possible parses showing how likely

each one is, or maybe to just return the most likely parse of a sentence. Thinking like this, the most natural thing to do is to define a probabilistic
PARSING MODEL *parsing model*, which evaluates the probability of trees t for a sentence s by finding:

$$P(t|s,G) \quad \text{where} \quad \sum_t P(t|s,G) = 1 \tag{12.3}$$

Given a probabilistic parsing model, the job of a parser is to find the most probable parse of a sentence $\hat{t}$:

$$\hat{t} = \arg\max_t P(t|s,G) \tag{12.4}$$

This is normally straightforward, but sometimes for practical reasons various sorts of heuristic or sampling parsers are used, methods which in most cases find the most probable parse, but sometimes don't.

One can directly estimate a parsing model, and people have done this, but they are a little odd in that one is using probabilities conditioned on a particular sentence. In general, we need to base our probability estimates on some more general class of data. The more usual approach
LANGUAGE MODEL is to start off by defining a language model, which assigns a probability to all trees generated by the grammar. Then we can examine the joint probability $P(t,s|G)$. Given that the sentence is determined by the tree (and recoverable from its leaf nodes), this is just $P(t|G)$, if $\text{yield}(t) = s$, and 0 otherwise. Under such a model, $P(t|G)$ is the probability of a particular parse of a particular sentence according to the grammar G. Below we suppress the conditioning of the probability according to the grammar, and just write $P(t)$ for this quantity.

In a language model, probabilities are for the entire language $\mathcal{L}$, so we have that:

$$\sum_{\{t:\ \text{yield}(t)\in\mathcal{L}\}} P(t) = 1 \tag{12.5}$$

We can find the overall probability of a sentence as:

$$\begin{aligned} P(s) &= \sum_t P(s,t) \\ &= \sum_{\{t:\ \text{yield}(t)=s\}} P(t) \end{aligned} \tag{12.6}$$

This means that it is straightforward to make a parsing model out of a language model. We simply divide the probability of a tree in the language model by the above quantity. The best parse is given by:

$$\hat{t} = \arg\max_t P(t|s) = \arg\max_t \frac{P(t,s)}{P(s)} = \arg\max_t P(t,s) \tag{12.7}$$

So a language model can always be used as a parsing model for the purpose of choosing between parses. But a language model can also be used for other purposes (for example, as a speech recognition language model, or for estimating the entropy of a language).

On the other hand, there is not a way to convert an arbitrary parsing model into a language model. Nevertheless, noticing some of the biases of PCFG parsing models that we discussed in chapter 11, a strand of work at IBM explored the idea that it might be better to build parsing models directly rather than defining them indirectly via a language model (Jelinek et al. 1994; Magerman 1995), and directly defined parsing models have also been used by others (Collins 1996). However, in this work, although the overall probabilities calculated are conditioned on a particular sentence, the atomic probabilities that the probability of a parse is decomposed into are not dependent on the individual sentence, but are still estimated from the whole training corpus. Moreover, when Collins (1997) refined his initial model (Collins 1996) so that parsing probabilities were defined via an explicit language model, this significantly increased the performance of his parser. So, while language models are not necessarily to be preferred to parsing models, they appear to provide a better foundation for modeling.

12.1.4 Weakening the independence assumptions of PCFGs

Context and independence assumptions

It is widely accepted in studies of language understanding that humans make wide use of the context of an utterance to disambiguate language as they listen. This use of context assumes many forms, for example the context where we are listening (to TV or in a bar), who we are listening to, and also the immediate prior context of the conversation. The prior discourse context will influence our interpretation of later sentences (this is the effect known as *priming* in the psychological literature). People will find semantically intuitive readings for sentences in preference to weird ones. Furthermore, much recent work shows that these many sources of

PRIMING

information are incorporated in real time while people parse sentences.[3] In our previous PCFG model, we were effectively making an independence assumption that none of these factors were relevant to the probability of a parse tree. But, in fact, all of these sources of evidence are relevant to and might be usable for disambiguating probabilistic parses. Even if we are not directly modeling the discourse context or its meaning, we can approximate these by using notions of collocation to help in more local semantic disambiguation, and the prior text as an indication of discourse context (for instance, we might detect the genre of the text, or its topic). To build a better statistical parser than a PCFG, we want to be able to incorporate at least some of these sources of information.

Lexicalization

LEXICALIZATION

There are two somewhat separable weaknesses that stem from the independence assumptions of PCFGs. The most often remarked on one is their lack of *lexicalization*. In a PCFG, the chance of a VP expanding as a verb followed by two noun phrases is independent of the choice of verb involved. This is ridiculous, as this possibility is much more likely with ditransitive verbs like *hand* or *tell*, than with other verbs. Table 12.2 uses data from the Penn Treebank to show how the probabilities of various common subcategorization frames differ depending on the verb that heads the VP.[4] This suggests that somehow we want to include more information about what the actual words in the sentence are when making decisions about the structure of the parse tree.

In other places as well, the need for lexicalization is obvious. A clear case is the issue of choosing phrasal attachment positions. As discussed at length in chapter 8, it is clear that the lexical content of phrases almost always provides enough information to decide the correct attachment site, whereas the syntactic category of the phrase normally provides very little information. One of the ways in which standard PCFGs are much

3. This last statement is not uncontroversial. Work in psycholinguistics that is influenced by a Chomskyan approach to language has long tried to argue that people construct syntactic parses first, and then choose between them in a disambiguation phase (e.g., Frazier 1978). But a variety of recent work (e.g., Tanenhaus and Trueswell 1995, Pearlmutter and MacDonald 1992) has argued against this and suggested that semantic and contextual information *does* get incorporated immediately during sentence understanding.

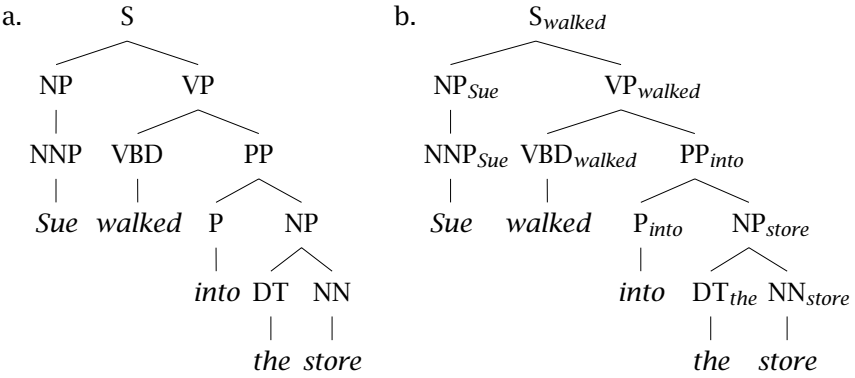
4. One can't help but suspect that some of the very low but non-zero entries might reveal errors in the treebank, but note that because functional tags are being ignored, an NP can appear after an intransitive verb if it is a temporal NP like *last week*.

	Verb			
Local tree	*come*	*take*	*think*	*want*
VP → V	9.5%	2.6%	4.6%	5.7%
VP → V NP	1.1%	32.1%	0.2%	13.9%
VP → V PP	34.5%	3.1%	7.1%	0.3%
VP → V SBAR	6.6%	0.3%	73.0%	0.2%
VP → V S	2.2%	1.3%	4.8%	70.8%
VP → V NP S	0.1%	5.7%	0.0%	0.3%
VP → V PRT NP	0.3%	5.8%	0.0%	0.0%
VP → V PRT PP	6.1%	1.5%	0.2%	0.0%

Table 12.2 Frequency of common subcategorization frames (local trees expanding VP) for selected verbs. The data show that the rule used to expand VP is highly dependent on the lexical identity of the verb. The counts ignore distinctions in verbal form tags. Phrase names are as in table 12.1, and tags are Penn Treebank tags (tables 4.5 and 4.6).

worse than *n*-gram models is that they totally fail to capture the lexical dependencies between words. We want to get this back, while maintaining a richer model than the purely linear word-level *n*-gram models. The most straightforward and common way to lexicalize a CFG is by having each phrasal node be marked by its head word, so that the tree in (12.8a) will be lexicalized as the tree in (12.8b).

(12.8)



Central to this model of lexicalization is the idea that the strong lexical dependencies are between heads and their dependents, for example between a head noun and a modifying adjective, or between a verb and

a noun phrase object, where the noun phrase object can in turn be approximated by its head noun. This is normally true and hence this is an effective strategy, but it is worth pointing out that there are some dependencies between pairs of non-heads. For example, for the object NP in (12.9):

(12.9) I got [$_{NP}$ the easier problem [of the two] [to solve]].

both the posthead modifiers *of the two* and *to solve* are dependents of the prehead modifier *easier*. Their appearance is only weakly conditioned by the head of the NP *problem*. Here are two other examples of this sort, where the head is in bold, and the words involved in the nonhead dependency are in italics:

(12.10) a. Her approach was *more quickly* **understood** *than mine*.

b. He lives in what must be the *farthest* **suburb** *from the university*.

See also exercise 8.16.

Probabilities dependent on structural context

However, PCFGs are also deficient on purely structural grounds. Inherent to the idea of a PCFG is that probabilities are context-free: for instance, that the probability of a noun phrase expanding in a certain way is independent of where the NP is in the tree. Even if we in some way lexicalize PCFGs to remove the other deficiency, this assumption of structural context-freeness remains. But this grammatical assumption is actually quite wrong. For example, table 12.3 shows how the probabilities of expanding an NP node in the Penn Treebank differ wildly between subject position and object position. Pronouns, proper names and definite NPs appear more commonly in subject position while NPs containing posthead modifiers and bare nouns occur more commonly in object position. This reflects the fact that the subject normally expresses the sentence-internal topic. As another example, table 12.4 compares the expansions for the first and second object NPs of ditransitive verbs. The dispreference for pronouns to be second objects is well-known, and the preference for 'NP SBAR' expansions as second objects reflects the well-known tendency for heavy elements to appear at the end of the clause, but it would take a more thorough corpus study to understand some of the other effects. For instance, it is not immediately clear to us why bare plural

Expansion	% as Subj	% as Obj
NP → PRP	13.7%	2.1%
NP → NNP	3.5%	0.9%
NP → DT NN	5.6%	4.6%
NP → NN	1.4%	2.8%
NP → NP SBAR	0.5%	2.6%
NP → NP PP	5.6%	14.1%

Table 12.3 Selected common expansions of NP as Subject vs. Object, ordered by log odds ratio. The data show that the rule used to expand NP is highly dependent on its parent node(s), which corresponds to either a subject or an object.

Expansion	% as 1st Obj	% as 2nd Obj
NP → NNS	7.5%	0.2%
NP → PRP	13.4%	0.9%
NP → NP PP	12.2%	14.4%
NP → DT NN	10.4%	13.3%
NP → NNP	4.5%	5.9%
NP → NN	3.9%	9.2%
NP → JJ NN	1.1%	10.4%
NP → NP SBAR	0.3%	5.1%

Table 12.4 Selected common expansions of NP as first and second object inside VP. The data are another example of the importance of structural context for nonterminal expansions.

nouns are so infrequent in the second object position. But at any rate, the context-dependent nature of the distribution is again manifest.

The upshot of these observations is that we should be able to build a much better probabilistic parser than one based on a PCFG by better taking into account lexical and structural context. The challenge (as so often) is to find factors that give us a lot of extra discrimination while not defeating us with a multiplicity of parameters that lead to sparse data problems. The systems in the second half of this chapter present a number of approaches along these lines.

(a)
S
NP VP
N VP
astronomers VP
astronomers V NP
astronomers saw NP
astronomers saw N
astronomers saw telescopes

(b)
S
NP VP
N VP
astronomers VP
astronomers V NP
astronomers V N
astronomers V *telescopes*
astronomers saw telescopes

Figure 12.3 Two CFG derivations of the same tree.

12.1.5 Tree probabilities and derivational probabilities

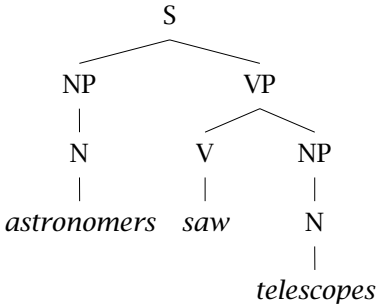
In the PCFG framework, one can work out the probability of a tree by just multiplying the probabilities of each local subtree of the tree, where the probability of a local subtree is given by the rule that produced it. The tree can be thought of as a compact record of a branching process where one is making a choice at each node, conditioned solely on the label of the node. As we saw in chapter 3, within generative models of syntax,[5] one generates sentences from a grammar, classically by starting with a start symbol, and performing a derivation which is a sequence of top-down rewrites until one has a phrase marker all of whose leaf nodes are terminals (that is, words). For example, figure 12.3 (a) shows the derivation of a sentence using the grammar of table 11.2, where at each stage one nonterminal symbol gets rewritten according to the grammar. A straightforward way to make rewrite systems probabilistic is to define probability distributions over each choice point in the derivation. For instance, at the last step, we chose to rewrite the final N as *telescopes*, but could have chosen something else, in accord with the grammar. The linear steps of a derivational process map directly onto a standard stochastic process, where the states are productions of the grammar. Since the generative grammar can generate all sentences of the language, a derivational model is inherently a language model.

Thus a way to work out a probability for a parse tree is in terms of the probability of derivations of it. Now in general a given parse tree can have multiple derivations. For instance, the tree in (12.11) has not

5. In the original sense of Chomsky (1957); in more recent work Chomsky has suggested that 'generative' means nothing more than 'formal' (Chomsky 1995: 162).

only the derivation in figure 12.3 (a), but also others, such as the one in figure 12.3 (b), where the second NP is rewritten before the V.

(12.11)



So, in general, to estimate the probability of a tree, we have to calculate:

(12.12) $$P(t) = \sum_{\{d:\ d \text{ is a derivation of } t\}} P(d)$$

However, in many cases, such as the PCFG case, this extra complication is unnecessary. It is fairly obvious to see (though rather more difficult to prove) that the choice of derivational order in the PCFG case makes no difference to the final probabilities.[6] Regardless of what probability distribution we assume over the choice of which node to rewrite next in a derivation, the final probability for a tree is otherwise the same. Thus we can simplify things by finding a way of choosing a unique derivation for each tree, which we will refer to as a *canonical derivation*. For instance, the leftmost derivation shown in figure 12.3 (a), where at each step we expand the leftmost non-terminal can be used as a canonical derivation. When this is possible, we can say:

CANONICAL DERIVATION

(12.13) $P(t) = P(d)$ where d is the canonical derivation of t

Whether this simplification is possible depends on the nature of the probabilistic conditioning in the model. It is possible in the PCFG case because probabilities depend only on the parent node, and so it doesn't matter if other nodes have been rewritten yet or not. If more context is used, or there are alternative ways to generate the same pieces of structure, then the probability of a tree might well depend on the derivation. See sections 12.2.1 and 12.2.2.[7]

6. The proof depends on using the kind of derivation to tree mapping developed in (Hopcroft and Ullman 1979).

7. Even in such cases, one might choose to approximate tree probabilities by estimating

Let us write $\alpha_u \overset{r_i}{\rightarrow} \alpha_v$ for an individual rewriting step r_i rewriting the string α_u as α_v. To calculate the probability of a derivation, we use the chain rule, and assign a probability to each step in the derivation, conditioned by preceding steps. For a standard rewrite grammar, this looks like this:

$$P(d) = P(\mathrm{S} \overset{r_1}{\rightarrow} \alpha_1 \overset{r_2}{\rightarrow} \alpha_2 \overset{r_3}{\rightarrow} \ldots \overset{r_m}{\rightarrow} \alpha_m = s) = \prod_{i=1}^{m} P(r_i | r_1, \ldots r_{i-1}) \tag{12.14}$$

We can think of the conditioning terms above, that is, the rewrite rules already applied, as the history of the parse, which we will refer to as h_i. So $h_i = (r_1, \ldots, r_{i-1})$. This is what led to the notion of *history-based grammars* (HBGs) explored initially at IBM. Since we can never model the entire history, normally what we have to do is form equivalence classes of the history via an equivalencing function π and estimate the above as:

HISTORY-BASED GRAMMARS

$$P(d) = \prod_{i=1}^{m} P(r_i | \pi(h_i)) \tag{12.15}$$

This framework includes PCFGs as a special case. The equivalencing function for PCFGs simply returns the leftmost non-terminal remaining in the phrase marker. So, $\pi(h_i) = \pi(h_i')$ iff $\mathrm{leftmost}_{\mathrm{NT}}(\alpha_i) = \mathrm{leftmost}_{\mathrm{NT}}(\alpha_i')$.

12.1.6 There's more than one way to do it

The way we augmented a CFG with probabilities in chapter 11 seems so natural that one might think that this is the only, or at least the only sensible, way to do it. The use of the term PCFG – probabilistic context-free grammar – tends to give credence to this view. Hence it is important to realize that this is untrue. Unlike the case of categorical context free languages, where so many different possibilities and parsing methods converge on strongly or weakly equivalent results, with probabilistic grammars, different ways of doing things normally lead to different probabilistic grammars. What is important from the probabilistic viewpoint is what the probabilities of different things are conditioned on (or looking from the other direction, what independence assumptions are made). While probabilistic grammars are sometimes equivalent – for example

them according to the probabilities of a canonical derivation, but this could be expected to have a detrimental effect on performance.

an HMM working from left-to-right gives the same results as one working from right-to-left, if the conditioning fundamentally changes, then there will be a different probabilistic grammar, even if it has the same categorical base. As an example of this, we will consider here another way of building a probabilistic grammar with a CFG basis, Probabilistic Left-Corner Grammars (PLCGs).

Probabilistic left-corner grammars

TOP-DOWN PARSING

If we think in parsing terms, a PCFG corresponds to a probabilistic version of *top-down parsing*. This is because at each stage we are trying to predict the child nodes given knowledge only of the parent node. Other parsing methods suggest different models of probabilistic conditioning. Usually, such conditioning is a *mixture of top-down and bottom-up information*. One such possibility is suggested by a left-corner parsing strategy.

LEFT CORNER PARSER

Left corner parsers (Rosenkrantz and Lewis 1970; Demers 1977) work by a combination of bottom-up and top-down processing. One begins with a goal category (the root of what is currently being constructed), and then looks at the left corner of the string (i.e., one shifts the next terminal). If the left corner is the same category as the goal category, then one can stop. Otherwise, one projects a possible local tree from the left corner, by looking for a rule in the grammar which has the left corner category as the first thing on its right hand side. The remaining children of this projected local tree then become goal categories and one recursively does left corner parsing of each. When this local tree is finished, one again recursively does left-corner parsing with the subtree as the left corner, and the same goal category as we started with. To make this description more precise, pseudocode for a simple left corner recognizer is shown in figure 12.4.[8] This particular parser assumes that lexical material is introduced on the right-hand side of a rule, e.g., as N → *house*, and that the top of the stack is to the left when written horizontally. The parser works in terms of a stack of found and sought constituents, the latter being represented on the stack as categories with a bar over them. We use α to represent a single terminal or non-terminal (or the empty string, if we wish to accommodate empty categories in the grammar), and γ to stand for a (possibly empty) sequence of terminals and

8. The presentation here borrows from an unpublished manuscript of Mark Johnson and Ed Stabler, 1993.

```
1  comment: Initialization
2  Place the predicted start symbol S̄ on top of the stack
3  comment: Parser
4  while (an action is possible) do one of the following
5        actions
6        [Shift] Put the next input symbol on top of the stack
7        [Attach] If ασ̄ is on top of the stack, remove both
8        [Project] If α is on top of the stack and A → α γ, replace α by γ̄A
9        endactions
10 end
11 comment: Termination
12 if empty(input) ∧ empty(stack)
13   then
14         exit success
15    else
16         exit failure
17 fi
```

Figure 12.4 An LC stack parser.

SHIFTING
PROJECTING
ATTACHING

non-terminals. The parser has three operations, *shifting*, *projecting*, and *attaching*. We will put probability distributions over these operations. When to shift is deterministic: If the thing on top of the stack is a sought category $\overline{C}$, then one must shift, and one can never successfully shift at other times. But there will be a probability distribution over what is shifted. At other times we must decide whether to attach or project. The only interesting choice here is deciding whether to attach in cases where the left corner category and the goal category are the same. Otherwise we must project. Finally we need probabilities for projecting a certain local tree given the left corner (lc) and the goal category (gc). Under this model, we might have probabilities like this:

$$\begin{aligned} P(\text{SBAR} \rightarrow \text{IN S} | lc = \text{IN}, gc = \text{S}) &= 0.25 \\ P(\text{PP} \rightarrow \text{IN NP} | lc = \text{IN}, gc = \text{S}) &= 0.55 \end{aligned}$$

To produce a language model that reflects the operation of a left corner parser, we can regard each step of the parsing operation as a step in a derivation. In other words, we can generate trees using left corner probabilities. Then, just as in the last section, we can express the probability of

a parse tree in terms of the probabilities of left corner derivations of that parse tree. Under left corner generation, each parse tree has a unique derivation and so we have:

$$P_{\text{lc}}(t) = P_{\text{lc}}(d) \quad \text{where } d \text{ is the LC derivation of } t$$

And the left corner probability of a sentence can then be calculated in the usual way:

$$P_{\text{lc}}(s) = \sum_{\{t:\ \text{yield}(t)=s\}} P_{\text{lc}}(t)$$

The probability of a derivation can be expressed as a product in terms of the probabilities of each of the individual operations in the derivation. Suppose that $(C_1, \ldots, C_m)$ is the sequence of operations in the LC parse derivation d of t. Then, by the chain rule, we have:

$$P(t) = P(d) = \prod_{i=1}^{m} P(C_i | C_1, \ldots, C_{i-1})$$

In practice, we cannot condition the probability of each parse decision on the entire history. The simplest left-corner model, which is all that we will develop here, assumes that the probability of each parse decision is largely independent of the parse history, and just depends on the state of the parser. In particular, we will assume that it depends simply on the left corner and top goal categories of the parse stack.

Each elementary operation of a left corner parser is either a shift, an attach or a left corner projection. Under the independence assumptions mentioned above, the probability of a shift will simply be the probability of a certain left corner child (lc) being shifted given the current goal category (gc), which we will model by P_{shift}. When to shift is deterministic. If a goal (i.e., barred) category is on top of the stack (and hence there is no left corner category), then one must shift. Otherwise one cannot. If one is not shifting, one must choose to attach or project, which we model by P_{att}. Attaching only has a non-zero probability if the left corner and the goal category are the same, but we define it for all pairs. If we do not attach, we project a constituent based on the left corner with probability P_{proj}. Thus the probability of each elementary operation C_i can be expressed in terms of probability distributions P_{shift}, P_{att}, and P_{proj} as follows:

$$P(C_i = \text{shift}\, lc) = \begin{cases} P_{shift}(lc|gc) & \text{if top is } gc \\ 0 & \text{otherwise} \end{cases} \tag{12.16}$$

$$P(C_i = \text{attach}) = \begin{cases} P_{att}(lc, gc) & \text{if top is not } gc \\ 0 & \text{otherwise} \end{cases} \tag{12.17}$$

$$P(C_i = \text{proj } A \rightarrow \gamma) = \begin{cases} (1 - P_{att}(lc, gc))P_{proj}(A \rightarrow \gamma | lc, gc) & \\ & \text{if top is not } gc \\ 0 & \text{otherwise} \end{cases} \tag{12.18}$$

Where these operations obey the following constraints:

$$\sum_{lc} P_{shift}(lc|gc) = 1 \tag{12.19}$$

$$\text{If } lc \neq gc, P_{att}(lc, gc) = 0 \tag{12.20}$$

$$\sum_{\{A \rightarrow \gamma:\ \gamma = lc\ \ldots\}} P_{proj}(A \rightarrow \gamma | lc, gc) = 1 \tag{12.21}$$

From the above we note that the probabilities of the choice of different shifts and projections sum to one, and hence, since other probabilities are complements of each other, the probabilities of the actions available for each elementary operation sum to one. There are also no dead ends in a derivation, because unless A is a possible left corner constituent of gc, $P_{proj}(A \rightarrow \gamma | lc, gc) = 0$. Thus we have shown that these probabilities define a language model.[9] That is, $\sum_s P_{\text{lc}}(s|G) = 1$.

Manning and Carpenter (1997) present some initial exploration of this form of PLCGs. While the independence assumptions used above are still quite drastic, one nevertheless gets a slightly richer probabilistic model than a PCFG, because elementary left-corner parsing actions are conditioned by the goal category, rather than simply being the probability of a local tree. For instance, the probability of a certain expansion of NP can be different in subject position and object position, because the goal category is different. So the distributional differences shown in table 12.3 can be captured.[10] Manning and Carpenter (1997) show how, because of this, a PLCG significantly outperforms a basic PCFG.

Other ways of doing it

Left-corner parsing is a particularly interesting case: left-corner parsers work incrementally from left-to-right, combine top-down and bottom-up prediction, and hold pride of place in the family of Generalized Left Corner Parsing models discussed in exercise 12.6. Nevertheless it is not the

9. Subject to showing that the probability mass accumulates in finite trees, the issue discussed in chapter 11.
10. However, one might note that those in table 12.4 will not be captured.

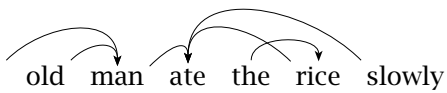
only other possibility for making probabilistic parsers based on CFG parsing algorithms, and indeed other approaches were investigated earlier.

Working with bottom-up shift-reduce parsers is another obvious possibility. In particular, a thread of work has looked at making probabilistic versions of the Generalized LR parsing approach of Tomita (1991). Briscoe and Carroll (1993) did the initial work in this area, but their model is probabilistically improper in that the LR parse tables guide a unification-based parser, and unification failures cause parse failures that are not captured by the probability distributions. A solidly probabilistic LR parser is described in (Inui et al. 1997).

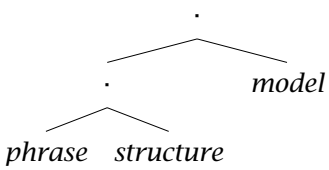
12.1.7 Phrase structure grammars and dependency grammars

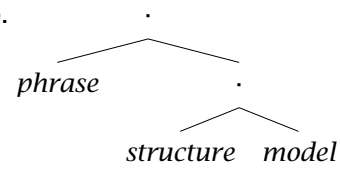
The dominant tradition within modern linguistics and NLP has been to use phrase structure trees to describe the structure of sentences. But an alternative, and much older, tradition is to describe linguistic structure in terms of dependencies between words. Such a framework is referred to as a *dependency grammar*. In a dependency grammar, one word is the head of a sentence, and all other words are either a dependent of that word, or else dependent on some other word which connects to the headword through a sequence of dependencies. Dependencies are usually shown as curved arrows, as for example in (12.22).

DEPENDENCY GRAMMAR

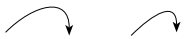
(12.22) 

Thinking in terms of dependencies is useful in Statistical NLP, but one also wants to understand the relationship between phrase structure and dependency models. In his work on disambiguating compound noun structures (see page 286), Lauer (1995a; 1995b) argues that a dependency model is better than an adjacency model. Suppose we want to disambiguate a compound noun such as *phrase structure model*. Previous work had considered the two possible tree structures for this compound noun, as shown in (12.23) and had tried to choose between them according to whether corpus evidence showed a tighter collocational bond between *phrase↔structure* or between *structure↔model*.

(12.23) a. 

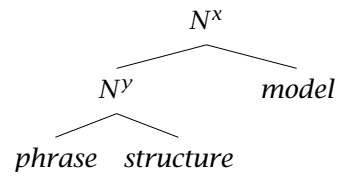
b. 

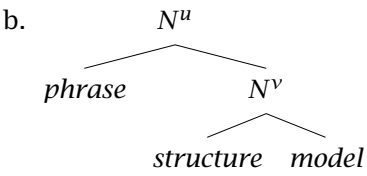
Lauer argues that instead one should examine the ambiguity in terms of dependency structures, as in (12.24), and there it is clear that the difference between them is whether *phrase* is a dependent of *structure* or whether it is a dependent of *model*. He tests this model against the adjacency model and shows that the dependency model outperforms the adjacency model.

(12.24) a. 

b. 

Now Lauer is right to point out that the earlier work had been flawed, and could maintain that it is easier to see what is going on in a dependency model. But this result does not show a fundamental advantage of dependency grammars over phrase structure grammars. The problem with the adjacency model was that in the trees, repeated annotated as (12.25), the model was only considering the nodes N^y and N^v, and ignoring the nodes N^x and N^u.

(12.25) a. 

b. 

If one corrects the adjacency model so that one also considers the nodes N^x and N^u, and does the obvious lexicalization of the phrase structure tree, so that N^y is annotated with *structure* and N^v with *model* (since English noun compounds are right-headed), then one can easily see that the two models become equivalent. Under a lexicalized PCFG type model, we find that $P(N^x) = P(N^v)$, and so the way to decide between the possibilities is by comparing $P(N^y)$ vs. $P(N^u)$. But this is exactly equivalent to comparing the bond between *phrase* → *structure* and *phrase* → *model*.

There are in fact isomorphisms between various kinds of dependency grammars and corresponding types of phrase structure grammars. A dependency grammar using undirected arcs is equivalent to a phrase structure grammar where every rule introduces at least one terminal node. For

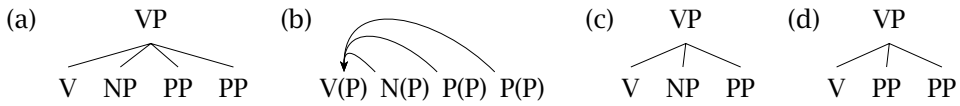


Figure 12.5 Decomposing a local tree into dependencies.

the more usual case of directed arcs, the equivalence is with 1-bar level X′ grammars. That is, for each terminal t in the grammar, there is a non-terminal $\overline{t}$, and the only rules in the grammar are of the form $\overline{t} \rightarrow \alpha\ t\ \beta$ where α and β are (possibly empty) sequences of non-terminals (cf. section 3.2.3). Another common option in dependency grammars is for the dependencies to be labeled. This in turn is equivalent to not only labeling one child of each local subtree as the *head* (as was implicitly achieved by the X-bar scheme), but labeling every child node with a relationship. Providing the probabilistic conditioning is the same, these results carry over to the probabilistic versions of both kinds of grammars.[11]

HEAD

Nevertheless, dependency grammars have their uses in probabilistic parsing, and, indeed, have become increasingly popular. There appear to be two key advantages. We argued before that lexical information is key to resolving most parsing ambiguities. Because dependency grammars work directly in terms of dependencies between words, disambiguation decisions are being made directly in terms of these word dependencies. There is no need to build a large superstructure (that is, a phrase structure tree) over a sentence, and there is no need to make disambiguation decisions high up in that structure, well away from the words of the sentence. In particular, there is no need to worry about questions of how to lexicalize a phrase structure tree, because there simply is no structure that is divorced from the words of the sentence. Indeed, a dependency grammarian would argue that much of the superstructure of a phrase structure tree is otiose: it is not really needed for constructing an understanding of sentences.

The second advantage of thinking in terms of dependencies is that dependencies give one a way of decomposing phrase structure rules, and estimates of their probabilities. A problem with inducing parsers from the Penn Treebank is that, because the trees are very flat, there are lots

11. Note that there is thus no way to represent within dependency grammars the two or even three level X′ schemata that have been widely used in modern phrase structure approaches.

of rare kinds of flat trees with many children. And in unseen data, one will encounter yet other such trees that one has never seen before. This is problematic for a PCFG which tries to estimate the probability of a local subtree all at once. Note then how a dependency grammar decomposes this, by estimating the probability of each head-dependent relationship separately. If we have never seen the local tree in figure 12.5 (a) before, then in a PCFG model we would at best back off to some default 'unseen tree' probability. But if we decompose the tree into dependencies, as in (b), then providing we had seen other trees like (c) and (d) before, then we would expect to be able to give quite a reasonable estimate for the probability of the tree in (a). This seems much more promising than simply backing off to an 'unseen tree' probability, but note that we are making a further important independence assumption. For example, here we might be presuming that the probability of a PP attaching to a VP (that is, a preposition depending on a verb in dependency grammar terms) is independent of how many NPs there are in the VP (that is, how many noun dependents the verb has). It turns out that assuming complete independence of dependencies does not work very well, and we also need some system to account for the relative ordering of dependencies. To solve these problems, practical systems adopt various methods of allowing some conditioning between dependencies (as described below).

12.1.8 Evaluation

An important question is how to evaluate the success of a statistical parser. If we are developing a language model (not just a parsing model), then one possibility is to measure the cross entropy of the model with respect to held out data. This would be impeccable if our goal had merely been to find some form of structure in the data that allowed us to predict the data better. But we suggested earlier that we wanted to build probabilistic parsers that found particular parse trees that we had in mind, and so, while perhaps of some use as an evaluation metric, ending up doing evaluation by means of measuring cross entropy is rather inconsistent with our stated objective. Cross entropy or perplexity measures only the probabilistic weak equivalence of models, and not the tree structure that we regard as important for other tasks. In particular, probabilistically weakly equivalent grammars have the same cross entropy, but if they are not strongly equivalent, we may greatly prefer one or the other for our task.

Why are we interested in particular parse trees for sentences? People are rarely interested in syntactic analysis for its own sake. Presumably our ultimate goal is to build a system for information extraction, question answering, translation, or whatever. In principle a better way to evaluate parsers is to embed them in such a larger system and to investigate the differences that the various parsers make in such a task-based evaluation. These are the kind of differences that someone outside the parsing community might actually care about.

However, often a desire for simplicity and modularization means that it would be convenient to have measures on which a parser can be simply and easily evaluated, and which one might expect to lead to better performance on tasks. If we have good reason to believe that a certain style of parse tree is useful for further tasks, then it seems that what we could do is compare the parses found by the program with the results of hand-parsing of sentences, which we regard as a gold standard. But how should we evaluate our parsing attempts, or in other words, what is the *objective criterion* that we are trying to maximize? The strictest criterion is to award the parser 1 point if it gets the parse tree completely right, and 0 points if it makes any kind of mistake. This is the *tree accuracy* or *exact match* criterion. It is the toughest standard, but in many ways it is a sensible one to use. In part this is because most standard parsing methods, such as the Viterbi algorithm for PCFGs try to maximize this quantity. So, since it is generally sensible for one's objective criterion to match what one's parser is maximizing, in a way using this criterion makes sense. However, clearly, in this line of reasoning, we are putting the cart before the horse. But for many potential tasks, partly right parses are not much use, and so it is a reasonable objective criterion. For example, things will not work very well in a database query system if one gets the scope of operators wrong, and it does not help much that the system got part of the parse tree right.

OBJECTIVE CRITERION

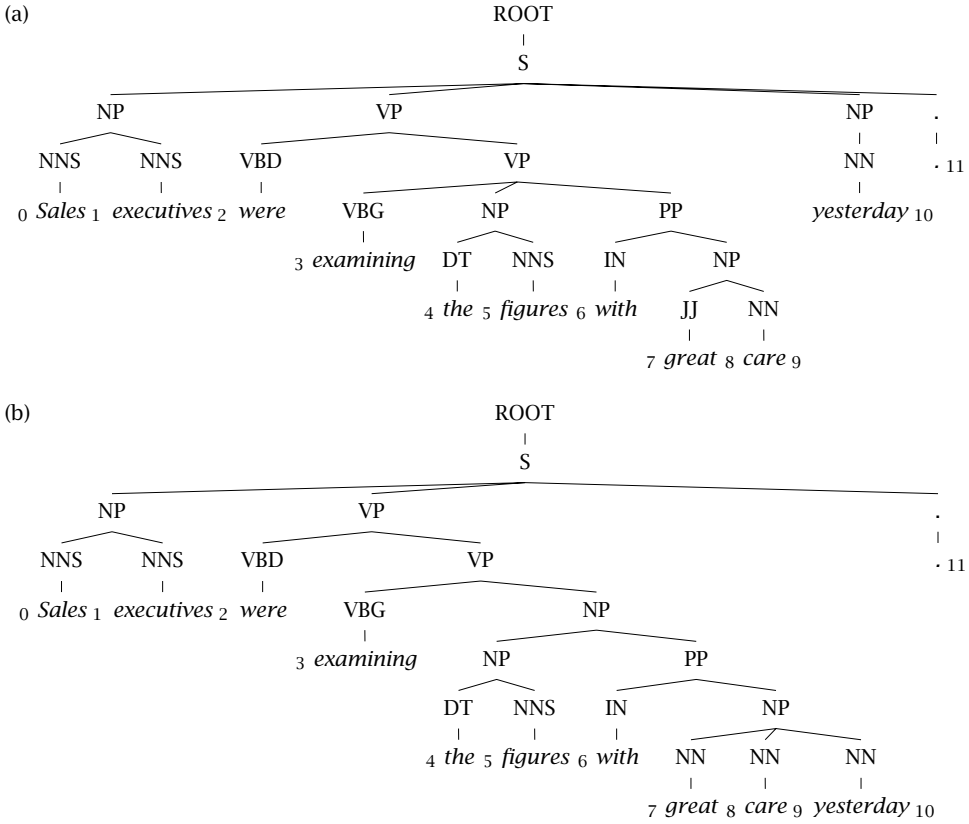
TREE ACCURACY

EXACT MATCH

On the other hand, parser designers, like students, appreciate getting part-credit for mostly right parses, and for some purposes partially right parses can be useful. At any rate, the measures that have most commonly been used for parser evaluation are the *PARSEVAL measures*, which originate in an attempt to compare the performance of non-statistical parsers. These measures evaluate the component pieces of a parse. An example of a parsed tree, a gold standard tree, and the results on the PARSEVAL measures as they have usually been applied in Statistical NLP work is shown in figure 12.6. Three basic measures are proposed: *precision* is

PARSEVAL MEASURES

PRECISION



(c) Brackets in gold standard tree (a.):
S-(0:11), **NP-(0:2)**, VP-(2:9), VP-(3:9), **NP-(4:6)**, PP-(6-9), NP-(7,9), *NP-(9:10)

(d) Brackets in candidate parse (b.):
S-(0:11), **NP-(0:2)**, VP-(2:10), VP-(3:10), NP-(4:10), **NP-(4:6)**, PP-(6-10), NP-(7,10)

(e)				
	Precision:	3/8 = 37.5%	Crossing Brackets:	0
	Recall:	3/8 = 37.5%	Crossing Accuracy:	100%
	Labeled Precision:	3/8 = 37.5%	Tagging Accuracy:	10/11 = 90.9%
	Labeled Recall:	3/8 = 37.5%		

Figure 12.6 An example of the PARSEVAL measures. The PARSEVAL measures are easily calculated by extracting the ranges which are spanned by non-terminal nodes, as indicated in (c) and (d) and then calculating the intersection, either including or not including labels while doing so. The matching brackets are shown in bold. The ROOT node is ignored in all calculations, and the preterminal nodes are used only for the tagging accuracy calculation. The starred unary node would be excluded in calculations according to the original standard, but is included here.

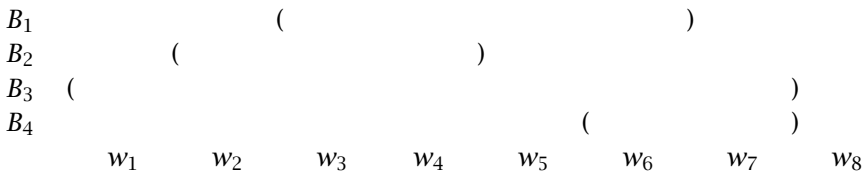


Figure 12.7 The idea of crossing brackets. Bracket B_1 crosses both brackets B_2 and B_4. All the other brackets are consistent. The guiding intuition is that crossing brackets cannot be combined into a single tree structure.

RECALL

CROSSING BRACKETS

how many brackets in the parse match those in the correct tree, *recall* measures how many of the brackets in the correct tree are in the parse, and *crossing brackets* gives the average of how many constituents in one tree cross over constituent boundaries in the other tree. A picture to help understand crossing brackets is in figure 12.7. Errors of this sort have often been seen as particularly dire. Rather than giving crossing brackets per tree (independent of tree size) an alternative is to report the non-crossing accuracy, the measure of the percentage of brackets that are not crossing brackets. The original PARSEVAL measures (because they were designed for implementations of various incompatible syntactic theories) ignored node labels, unary branching nodes, and performed various other (ad hoc) tree normalizations. However, if a parser is trying to duplicate parses being performed by hand, then it is reasonable to include node labels, and this is normally done, giving measures of labeled precision and recall. It is also reasonable to include unary nodes, except that the unary ROOT and preterminal nodes are not included.[12]

The PARSEVAL measures are not very discriminating. As we will see

12. The Penn Treebank gives all trees an unlabeled top node, here called ROOT. This is useful so that one can have rules rewriting the ROOT as the actual top node of the sentence, whether S or NP or whatever. (A surprising number of 'sentences' in newswire text are actually just noun phrases.) Including this node, and the preterminal nodes would inflate the precision and recall figures, as they are unary nodes which one could not get wrong. Including preterminal labels in the labeled measures is more defensible, but would mean that one's ability to do part of speech tagging is rolled into the performance of the parser, whereas some people feel it is best reported separately. Chains of unary nodes present some problems for these measures: the measures fail to capture the dominance ordering of the nodes, and, if multiple nodes have the same category, care must be taken in the calculation of recall. Finally we note that most evaluations of Statistical NLP parsers on the Penn Treebank have used their own ad-hoc normalizations, principally discounting sentence-internal punctuation and the distinction between the categories ADVP and PRT.

below, Charniak (1996) shows that according to these measures, one can do surprisingly well on parsing the Penn Treebank by inducing a vanilla PCFG which ignores all lexical content. This somewhat surprising result seems to reflect that in many respects the PARSEVAL measures are quite easy ones to do well on, particularly for the kind of tree structures assumed by the Penn Treebank. Firstly, it is important to note that they are measuring success at the level of individual decisions – and normally what makes NLP hard is that you have to make many consecutive decisions correctly to succeed. The overall success rate is then the n^{th} power of the individual decision success rate – a number that easily becomes small.

But beyond this, there are a number of features particular to the structure of the Penn Treebank that make these measures particularly easy. Success on crossing brackets is helped by the fact that Penn Treebank trees are quite flat. To the extent that sentences have very few brackets in them, the number of crossing brackets is likely to be small. Identifying troublesome brackets that would lower precision and recall measures is also avoided. For example, recall that there is no disambiguation of compound noun structures within the Penn Treebank, which gives a completely flat structure to a noun compound (and any other prehead modifiers) as shown below (note that the first example also illustrates the rather questionable Penn Treebank practice of tagging hyphenated non-final portions of noun compounds as adjectives!).

(12.26) [NP a/DT stock-index/JJ arbitrage/NN sell/NN program/NN]
[NP a/DT joint/JJ venture/NN advertising/NN agency/NN]

Another case where peculiarities of the Penn Treebank help is the non-standard adjunction structures given to post noun-head modifiers, of the general form (NP (NP the man) (PP in (NP the moon))). As we discussed in section 8.3, a frequent parsing ambiguity is whether PPs attach to a preceding NP or VP – or even to a higher preceding node – and this is a situation where lexical or contextual information is more important than structural factors. Note now that the use of the above adjunction structure reduces the penalty for making this decision wrongly. For the different tree brackettings for Penn Treebank style structures and the type of N′ structure more commonly assumed in linguistics, as shown in figure 12.8, the errors assessed for different attachments are as shown in table 12.5. The forgivingness of the Penn Treebank scheme is manifest.

Penn VP attach	(VP saw (NP the man) (PP with (NP a telescope)))
Penn NP attach	(VP saw (NP (NP the man) (PP with (NP a telescope))))
Another VP attach	(VP saw (NP the (N′ man)) (PP with (NP a (N′ telescope)))))
Another NP attach	(VP saw (NP the (N′ man (PP with (NP a (N′ telescope))))))

Figure 12.8 Penn trees versus other trees.

	Error	Errors assessed		
		Prec.	Rec.	CBs
Penn	VP instead of NP	0	1	0
	NP instead of VP	1	0	0
Another	VP instead of NP	2	2	1
	NP instead of VP	2	2	1

Table 12.5 Precision and recall evaluation results for PP attachment errors for different styles of phrase structure.

One can get the attachment wrong and not have any crossing brackets, and the errors in precision and recall are minimal.[13]

On the other hand, there is at least one respect in which the PARSEVAL measures seem too harsh. If there is a constituent that attaches very high (in a complex right-branching sentence), but the parser by mistake attaches it very low, then *every* node in the right-branching complex will be wrong, seriously damaging both precision and recall, whereas arguably only a single mistake was made by the parser. This is what happened to give the very bad results in figure 12.6. While there are two attachment errors in the candidate parse, the one that causes enormous damage in the results is attaching *yesterday* low rather than high (the parser which generated this example didn't know about temporal nouns, to its great detriment).

This all suggests that these measures are imperfect, and one might wonder whether something else should be introduced to replace them. One idea would be to look at dependencies, and to measure how many of the dependencies in the sentence are right or wrong. However, the difficulty in doing this is that dependency information is not shown in

13. This comparison assumes that one is including unary brackets. The general contrast remains even if one does not do so, but the badness of the non-Penn case is slightly reduced.

the Penn Treebank. While one can fairly successfully induce dependency relationships from the phrase structure trees given, there is no real gold standard available.

Returning to the idea of evaluating a parser with respect to a task, the correct approach is to examine whether success on the PARSEVAL measures is indicative of success on real tasks. Many small parsing mistakes might not affect tasks of semantic interpretation. This is suggested by results of (Bonnema 1996; Bonnema et al. 1997). For instance, in one experiment, the percentage of correct semantic interpretations was 88%, even though the tree accuracy of the parser was only 62%. The correlation between the PARSEVAL measures and task-based performance is briefly investigated by Hermjakob and Mooney (1997) with respect to their task of English to German translation. In general they find a quite good correlation between the PARSEVAL measures and generating acceptable translations. Labeled precision has by far the best correlation with a semantically adequate translation (0.78), whereas the correlation with the weaker measure of crossing brackets is much more modest (0.54). Whether there are other evaluation criteria that correlate better with success on final tasks, and whether different criteria better predict performance on different kinds of final tasks remain open questions. However, at the moment, people generally feel that these measures are adequate for the purpose of comparing parsers.

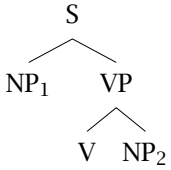
12.1.9 Equivalent models

When comparing two probabilistic grammars, it is easy to think that they are different because they are using different surface trappings, but what is essential is to work out what information is being used to condition the prediction of what. Providing the answers to that question are the same, then the probabilistic models are equivalent.

In particular, often there are three different ways of thinking about things: in terms of remembering more of the derivational history, looking at a bigger context in a phrase structure tree, or by enriching the vocabulary of the tree in deterministic ways.

Let us take a simple example. Johnson (1998) demonstrates the utility of using the grandparent node (G) as additional contextual information when rewriting a parent non-terminal (P) in a PCFG. For instance, consider the tree in (12.27).

(12.27)



When expanding the NP non-terminals in (12.27), for NP_1, we would be using $P(NP \rightarrow \alpha | \mathcal{P} = NP, \mathcal{G} = S)$, while for NP_2 we would use $P(NP \rightarrow \alpha | \mathcal{P} = NP, \mathcal{G} = VP)$. This model can also capture the differences in the probability distributions for subject and object NPs shown in table 12.3 (while again failing to capture the distributional differences shown in table 12.4). Including information about the grandparent is surprisingly effective. Johnson shows that this simple model actually outperforms the probabilistic left-corner model presented earlier, and that in general it appears to be the most valuable simple enrichment of a PCFG model, short of lexicalization, and the concomitant need to handle sparse data that that introduces.

But the point that we wish to make here is that one can think of this model in three different ways: as using more of the derivational history, as using more of the parse tree context, or as enriching the category labels. The first way to think of it is in derivational terms, as in a history-based grammar. There we would be saying that we are doing a finer equivalence classing of derivational histories. For two derivational histories to be equivalent, not only would they have to have the same leftmost non-terminal remaining in the phrase marker, but both of these would have to have resulted from rewriting the same category. That is:

$$\pi(h) = \pi(h') \text{ iff } \begin{cases} \text{leftmost}_{NT}(\alpha_m) = \text{leftmost}_{NT}(\alpha'_m) = N^x \ \& \\ \exists N^y : N^y \rightarrow \ldots\ N^x\ \ldots \in h \wedge N^y \rightarrow \ldots\ N^x\ \ldots \in h' \end{cases}$$

If two non-terminals were in different equivalence classes, they would be able to (and usually would) have different probabilities for rewriting.

But, instead of doing this, we could think of this new model simply in terms of the probability of tree structures, but suggest that rather than working out the probability of a local subtree just by looking at the nodes that comprise the subtree, we could also look at more surrounding context. One can get into trouble if one tries to look at the surrounding context in all directions at once, because then one can no longer produce a well-founded probabilistic model or parsing method – there has to be a certain directionality in the use of context. But if one is thinking of the tree being built top-down, then one can certainly include as much context

from higher up in the tree as one wishes. Building equivalence classes of sequences of derivational steps is equivalent to building equivalence classes of partial trees. Just including the identity of the grandparent node is a particularly simple example of enriching context in this way.

Or thirdly, one can do what Johnson actually did and just use a generic PCFG parser, but enrich the vocabulary of the tree labels to encode this extra contextual information. Johnson simply relabeled every non-terminal with a composite label that recorded both the node's original label and its parent's label (for instance, NP_1 in (12.27) was relabeled as NP-S). Two nodes in the new trees had the same label if and only if both they and their parents had the same label in the original trees. Johnson could then use a standard PCFG parser over these new trees to simulate the effect of using extra contextual information in the original trees. All three of these methods produce equivalent probabilistic models. But the third method seems a particularly good one to remember, since it is frequently easier to write a quick program to produce transformed trees than to write a new probabilistic parser.

12.1.10 Building parsers: Search methods

For certain classes of probabilistic grammars, there are efficient algorithms that can find the highest probability parse in polynomial time. The way such algorithms work is by maintaining some form of *tableau* that stores steps in a parse derivation as they are calculated in a bottom-up fashion. The tableau is organized in such a way that if two subderivations are placed into one cell of the tableau, we know that both of them will be able to be extended in the same ways into larger subderivations and complete derivations. In such derivations, the lower probability one of the two will always lead to lower probability complete derivations, and so it may be discarded. Such algorithms are in general known as *Viterbi algorithms*, and we have seen a couple of examples in earlier chapters.

TABLEAU

VITERBI ALGORITHM

When using more complex statistical grammar formalisms, such algorithms may not be available. This can be for two reasons. There may not be (known) tabular parsing methods for these formalisms. But secondly, the above discussion assumed that by caching derivation probabilities one could efficiently find parse probabilities. Viterbi algorithms are a means of finding the highest probability derivation of a tree. They only allow us to find the highest probability parse for a tree if we can define a unique canonical derivation for each parse tree (as discussed earlier).

If there is not a one-to-one relationship between derivations and parses, then an efficient polynomial time algorithm for finding the highest probability tree may not exist. We will see an example below in section 12.2.1.

For such models, "the decoding problem" of finding the best parse becomes exponential. We nevertheless need some efficient way of moving through a large search space. If we think of a parsing problem as a search problem in this way, we can use any of the general search methods that have been developed within AI. But we will start with the original and best-known algorithm for doing this within the Statistical NLP community, the stack decoding algorithm.

The stack decoding algorithm

The stack decoding algorithm was initially developed by Jelinek (1969) for the purpose of decoding information transmissions across noisy channels. However, it is a method for exploring any tree-structured search space, such as commonly occurs in Statistical NLP algorithms. For example, a derivational parsing model gives a tree-structured search space, since we start with various choices for the first step of the derivation, and each of those will lead to a (normally different) range of choices for the second step of the derivation. It is an example of what in AI is known as a *uniform-cost search* algorithm: one where one always expands the least-cost leaf node first.

UNIFORM-COST SEARCH

The stack decoding algorithm can be described via a priority queue object, an ordered list of items with operations of pushing an item and popping the highest-ranked item. Priority queues can be efficiently implemented using a heap data structure.[14] One starts with a priority queue that contains one item – the initial state of the parser. Then one goes into a loop where at each step one takes the highest probability item off the top of the priority queue, and extends it by advancing it from an n step derivation to an $n + 1$ step derivation (in general there will be multiple ways of doing this). These longer derivations are placed back on the priority queue ordered by probability. This process repeats until there is a complete derivation on top of the priority queue. If one assumes an infinite priority queue, then this algorithm is guaranteed to find the highest probability parse, because a higher probability partial derivation will always be extended before a lower probability one. That is, it is *complete*

14. This is described in many books on algorithms, such as (Cormen et al. 1990).

(guaranteed to find a solution if there is one) and *optimal* (guaranteed to find the best solution when there are several). If, as is common, a limited priority queue size is assumed, then one is not guaranteed to find the best parse, but the method is an effective heuristic for usually finding the best parse. The term *beam search* is used to describe systems which only keep and extend the best partial results. A *beam* may either be fixed size, or keep all results whose goodness is within a factor α of the goodness of the best item in the beam.

BEAM SEARCH

In the simplest version of the method, as described above, when one takes the highest probability item off the heap, one finds all the possible ways to extend it from an n step derivation to an $n+1$ step derivation, by seeing which next parsing steps are appropriate, and pushing the resulting $n+1$ step derivations back onto the heap. But Jelinek (1969) describes an optimization (which he attributes to John Cocke), where instead of doing that, one only applies the highest probability next step, and therefore pushes only the highest probability $n+1$ step derivation onto the stack, together with continuation information which can serve to point to the state at step n and the other extensions that were possible. Thereafter, if this state is popped from the stack, one not only determines and pushes on the highest probability $n+2$ step derivation, but one retrieves the continuation, applies the second highest probability rule, and pushes on the second highest probability $n+1$ step derivation (perhaps with its own continuation). This method of working with continuations is in practice very effective at reducing the beam size needed for effective parsing using the stack decoding algorithm.

A* search

Uniform-cost search can be rather inefficient, because it will expand all partial derivations (in a breadth-first-like manner) a certain distance, rather than directly considering whether they are likely to lead to a high probability complete derivation. There exist also *best-first search* algorithms which do the opposite, and judge which derivation to expand based on how near to a complete solution it is. But really what we want to do is find a method that combines both of these and so tries to expand the derivation that looks like it will lead to the highest probability parse, based on both the derivational steps already taken and the work still left to do. Working out the probability of the steps already taken is easy. The tricky part is working out the probability of the work still to do. It turns

BEST-FIRST SEARCH

out, though, that the right thing to do is to choose an optimistic estimate, meaning that the probability estimate for the steps still to be taken is always equal to or higher than the actual cost will turn out to be. If we can do that, it can be shown that the resulting search algorithm is still complete and optimal. Search methods that work in this way are called *A* search* algorithms. A* search algorithms are much more efficient because they direct the parser towards the partial derivations that look nearest to leading to a complete derivation. Indeed, A* search is *optimally efficient* meaning that no other optimal algorithm can be guaranteed to explore less of the search space.

A* SEARCH

OPTIMALLY EFFICIENT

Other methods

We have merely scratched the surface of the literature on search methods. More information can be found in most AI textbooks, for example (Russell and Norvig 1995: ch. 3–4).

We might end this subsection by noting that in cases where the Viterbi algorithm is inapplicable, one also usually gives up 'efficient' training: one cannot use the EM algorithm any more either. But one can do other things. One approach which has been explored at IBM is growing a decision tree to maximize the likelihood of a treebank (see section 12.2.2).

12.1.11 Use of the geometric mean

Any standard probabilistic approach ends up multiplying a large number of probabilities. This sequence of multiplications is justified by the chain rule, but most usually, large assumptions of conditional independence are made to make the models usable. Since these independence assumptions are often quite unjustifiable, large errors may accumulate. In particular, failing to model dependencies tends to mean that the estimated probability of a tree becomes far too low. Two other problems are sparse data where probability estimates for infrequent unseen constructs may also be far too low, and defective models like PCFGs that are wrongly biased to give short sentences higher probabilities than long sentences. As a result of this, sentences with bigger trees, or longer derivational histories tend to be penalized in existing statistical parsers. To handle this, it has sometimes been suggested (Magerman and Marcus 1991; Carroll 1994) that one should rather calculate the geometric mean (or equivalently the average log probability) of the various derivational steps. Such

a move takes one out of the world of probabilistic approaches (however crude the assumptions) and into the world of *ad hoc* scoring functions for parsers. This approach can sometimes prove quite effective in practice, but it is treating the symptoms not the cause of the problem. For the goal of speeding up chart parsing, Caraballo and Charniak (1998) show that using the geometric mean of the probability of the rules making up a constituent works much better than simply using the probability of the constituent for rating which edges to focus on extending – this is both because the PCFG model is strongly biased to give higher probabilities to smaller trees, and because this measure ignores the probability of the rest of the tree. But they go on to show that one can do much better still by developing better probabilistic metrics of goodness.

12.2 Some Approaches

In the remainder of this chapter, we examine ways that some of the ideas presented above have been combined into statistical parsers. The presentations are quite brief, but give an overview of some of the methods that are being used and the current state of the art.

12.2.1 Non-lexicalized treebank grammars

A basic division in probabilistic parsers is between lexicalized parsers which deal with words, and those that operate over word categories. We will first describe non-lexicalized parsers. For a non-lexicalized parser, the input 'sentence' to parse is really just a list of word category tags, the preterminals of a normal parse tree. This obviously gives one much less information to go on than a sentence with real words, and in the second half we will discuss higher-performing lexicalized parsers. However, apart from general theoretical interest, the nice thing about non-lexicalized parsers is that the small terminal alphabet makes them easy to build. One doesn't have to worry too much about either computational efficiency or issues of smoothing sparse data.

PCFG estimation from a treebank: Charniak (1996)

Charniak (1996) addresses the important empirical question of how well a parser can do if it ignores lexical information. He takes the Penn Treebank, uses the part of speech and phrasal categories it uses (ignoring

functional tags), induces a maximum likelihood PCFG from the trees by using the relative frequency of local trees as the estimates for rules in the obvious way, makes no attempt to do any smoothing or collapsing of rules, and sets out to try to parse unseen sentences.[15]

The result was that this grammar performed surprisingly well. Its performance in terms of precision, recall, and crossing brackets is not far below that of the best lexicalized parsers (see table 12.6). It is interesting to consider why this is. This result is surprising because such a parser will always choose the same resolution of an attachment ambiguity when confronted with the same structural context – and hence must often be wrong (cf. section 8.3). We feel that part of the answer is that these scoring measures are undiscerning on Penn Treebank trees, as we discussed in section 12.1.8. But it perhaps also suggests that while interesting parsing decisions, such as classic attachment ambiguities, clearly require semantic or lexical information, perhaps the majority of parsing decisions are mundane, and can be handled quite well by an unlexicalized PCFG. The precision, recall, and crossing brackets measures record average performance, and one can fare quite well on average with just a PCFG.

The other interesting point is that this result was achieved without any smoothing of the induced grammar, despite the fact that the Penn Treebank is well-known for its flat many-branching constituents, many of which are individually rare. As Charniak shows, the grammar induced from the Penn Treebank ends up placing almost no categorical constraints on what part of speech can occur next in a sentence, so one can parse any sentence. While it is certainly true that some rare local trees appear in the test set that were unseen during training, it is unlikely that they would ever occur in the highest probability parse, even if smoothing were done. Thus, under these circumstances, just using maximum likelihood estimates does no harm.

Partially unsupervised learning: Pereira and Schabes (1992)

We have discussed how the parameter estimation space for realistic-sized PCFGs is so big that the EM algorithm unaided tends to be of fairly little use, because it always gets stuck in a local maximum. One way to try to

15. We simplify slightly. Charniak did do a couple of things: recoding auxiliary verbs via an AUX tag, and incorporating a 'right-branching correction,' so as to get the parser to prefer right branching structures.

encourage the probabilities into a good region of the parameter space is proposed by Pereira and Schabes (1992) and Schabes et al. (1993). They begin with a Chomsky normal form grammar with 15 non-terminals over an alphabet of 45 part of speech tags as terminals, and train it not on raw sentences but on treebank sentences, where they ignore the non-terminal labels, but use the treebank bracketing. They employ a variant of the Inside-Outside algorithm constrained so as to only consider parses that do not cross Penn-Treebank nodes. Their parser always parses into binary constituents, but it can learn from any style of bracketing, which the parser regards as a partial bracketing of the sentence. We will not present here their modified versions of the Inside-Outside algorithm equations, but the basic idea is to reduce to zero the contribution to the reestimation equations of any proposed constituent which is not consistent with the treebank bracketing. Since bracketing decreases the number of rule split points to be considered, a bracketed training corpus also speeds up the Inside-Outside algorithm.

On a small test corpus, Pereira and Schabes (1992) show the efficacy of the basic method. Interestingly, both the grammars trained on unbracketed and bracketed training material converge on a very similar cross-entropy, but they differ hugely on how well their bracketings correspond to the desired bracketings present in the treebank. When the input was unbracketed, only 37% of the brackets the parser put on test sentences were correct, but when it had been trained on bracketed sentences, 90% of the brackets placed on test sentences were correct. Moreover, while EM training on the unbracketed data was successful in decreasing the cross-entropy, it was ineffective at improving the bracketing accuracy of the parser over the accuracy of the model resulting from random initialization of the parameters. This result underlines the discussion at the beginning of the chapter: current learning methods are effective at finding models with low entropy, but they are insufficient to learn syntactic structure from raw text. Only by chance will the inferred grammar agree with the usual judgements of sentence structure. At the present time, it is an open question whether the normally assumed hierarchical structure of language is underdetermined by the raw data, or whether the evidence for it is simply too subtle to be discovered by current induction techniques.

Schabes et al. (1993) test the same method on a larger corpus including longer sentences with similar results. They make use of one additional interesting idea, which is to impose a uniform right branching binary

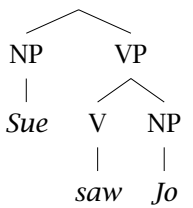
structure on all flat *n*-ary branching local trees of the Penn Treebank in the training data so as to maximize the speed-up to the Inside-Outside algorithm that comes from bracketing being present.

Parsing directly from trees: Data-Oriented Parsing

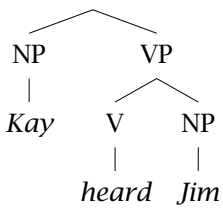
An interesting alternative to the grammar-based models that we have considered so far is to work out statistics directly on pieces of trees in a treebank, where the treebank is assumed to represent the body of parses that one has previously explored. Rather than deriving a grammar from the treebank, we let the parsing process use whichever fragments of trees appear to be useful. This has the apparent advantage that idiom chunks like *to take advantage of* will be used where they are present, whereas such chunks are not straightforwardly captured in PCFG-style models. Such an approach has been explored within the Data-Oriented Parsing (DOP) framework of Rens Bod and Remko Scha (Sima'an et al. 1994; Bod 1995, 1996, 1998). In this section, we will look at the DOP1 model.

Suppose we have a corpus of two sentences, as in (12.28):

(12.28) a.

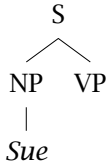


b.

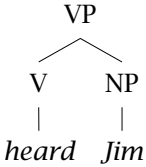


Then, to parse a new sentence like *Sue heard Jim*, we could do it by putting together tree fragments that we have already seen. For example we can compose these two tree fragments:

(12.29) a.



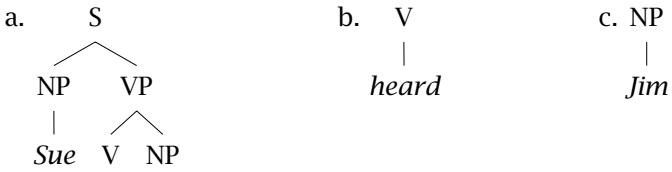
b.



We can work out the probability of each tree fragment in the corpus, given that one is expanding a certain node, and, assuming independence, we can multiply these probabilities together (for instance, there are 8

fragments with VP as the parent node – fragments must include either all or none of the children of a node – among which (12.29b) occurs once, so its probability is 1/8). But that is only one derivation of this parse tree. In general there are many. Here is another one from our corpus, this time involving the composition of three tree fragments:

(12.30)



Since there are multiple fundamentally distinct derivations of a single tree in this DOP model, here we have an example of a grammar where the highest probability parse cannot be found efficiently by a Viterbi algorithm (Sima'an 1996) – see exercise 12.8. Parsing has therefore been done using *Monte Carlo simulation* methods. This is a technique whereby the probability of an event is estimated by taking random samples. One randomly produces a large number of derivations and uses these to estimate the most probable parse. With a large enough sample, these estimates can be made as accurate as desired, but the parsing process becomes slow.

MONTE CARLO SIMULATION

The DOP approach is in some ways similar to the memory-based learning (MBL) approach (Zavrel and Daelemans 1997) in doing prediction directly from a corpus, but differs in that whereas the MBL approach predicts based on a few similar exemplars, the DOP model uses statistics over the entire corpus.

The DOP model provides a different way of thinking, but it is important to realize that it is not *that* different to what we have been doing with PCFGs. After all, rather than writing grammar rules like S → NP VP and VP → V NP, we could instead write tree fragments:

(12.31)

And the probabilities that we estimate for grammar rules from a treebank are exactly the same as would be assigned based on their relative frequency in the treebank on the DOP model.

The difference between PCFGs and what we have here is that rather than only having local trees of depth 1, we can have bigger tree frag-

PROBABILISTIC TREE SUBSTITUTION GRAMMAR

ments. The model can be formalized as a *Probabilistic Tree Substitution Grammar* (PTSG), which has five components just like the definition of a PCFG in chapter 11. However, rather than a set of rules, we have a set of tree fragments of arbitrary depth whose top and interior nodes are nonterminals and whose leaf nodes are terminals or nonterminals, and the probability function assigns probabilities to these fragments. PTSGs are thus a generalization of PCFGs, and are stochastically more powerful, because one can give particular probabilities to fragments – or just whole parses – which cannot be generated as a multiplication of rule probabilities in a PCFG. Bod (1995) shows that by starting with a PCFG model of depth 1 fragments and then progressively allowing in larger fragments parsing accuracy does increase significantly (this mirrors the result of Johnson (1998) on the utility of context from higher nodes in the tree). So the DOP model provides another way to build probabilistic models that use more conditioning context.

12.2.2 Lexicalized models using derivational histories

History-based grammars (HBGs)

Probabilistic methods based on the history of the derivation, and including a rich supply of lexical and other information, were first explored in large scale experiments at IBM, and are reported in (Black et al. 1993). This work exploited a one-to-one correspondence between leftmost derivations and parse trees, to avoid summing over possible derivations. The general idea was that all prior parse decisions could influence following parse decisions in the derivation, however, in the 1993 model, the only conditioning features considered were those on a path from the node currently being expanded to the root of the derivation, along with what number child of the parent a node is (from left to right).[16] Black et al. (1993) used decision trees to decide which features in the derivational history were important in determining the expansion of the current node. We will cover decision trees in section 16.1, but they can be thought of just as a tool that divides up the history into highly predictive equivalence classes.

16. Simply using a feature of being the n^{th} child of the parent seems linguistically somewhat unpromising, since who knows what material may be in the other children, but this gives some handle on the varying distribution shown in table 12.4.

Unlike most other work, this work used a custom treebank, produced by the University of Lancaster. In the 1993 experiments, they restricted themselves to sentences completely covered by the most frequent 3000 words in the corpus (which effectively avoids many sparse data issues). Black et al. began with an existing hand-built broad-coverage feature-based unification grammar. This was converted into a PCFG by making equivalence classes out of certain labels (by ignoring or grouping certain features and feature-value pairs). This PCFG was then reestimated using a version of the Inside-Outside algorithm that prevents bracket crossing, as in the work of Pereira and Schabes (1992) discussed above.

Black et al. lexicalize their grammar so that phrasal nodes inherit two words, a lexical head H_1, and a secondary head H_2. The lexical head is the familiar syntactic head of the phrase, while the secondary head is another word that is deemed useful (for instance, in a prepositional phrase, the lexical head is the preposition, while the secondary head is the head of the complement noun phrase). Further, they define a set of about 50 each of syntactic and semantic categories, $\{Syn_p\}$ and $\{Sem_p\}$, to be used to classify non-terminal nodes. In the HBG parser, these two features, the two lexical heads, and the rule R to be applied at a node are predicted based on the same features of the parent node, and the index I expressing what number child of the parent node is being expanded. That is, we wish to calculate:

$$P(Syn, Sem, R, H_1, H_2 | Syn_p, Sem_p, R_p, I_{pc}, H_{1p}, H_{2p})$$

This joint probability is decomposed via the chain rule and each of the features is estimated individually using decision trees.

The idea guiding the IBM work was that rather than having a linguist tinker with a grammar to improve parsing preferences, the linguist should instead just produce a parser that is capable of parsing all sentences. One then gets a statistical parser to learn from the information in a treebank so that it can predict the correct parse by conditioning parsing steps on the derivation history. The HBG parser was tested on sentences of 7–17 words, by comparing its performance to the existing unification-based parser. The unification-based parser chose the correct parse for sentences about 60% of the time, while the HBG parser found the correct parse about 75% of the time, so the statistical parser was successful in producing a 37% reduction in error over the best disambiguation rules that the IBM linguist had produced by hand.

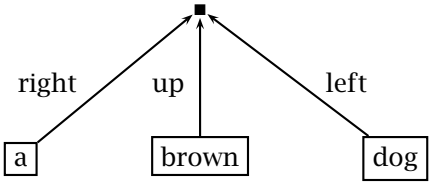
SPATTER

The HBG work was based on a language model, but work at IBM then started experimenting with building a parsing model directly. The early work reported in Jelinek et al. (1994) was developed as the SPATTER model in Magerman (1994, 1995), which we briefly review here.

SPATTER also works by determining probabilities over derivations, but it works in a bottom-up fashion, by starting with the words and building structure over them. Decision tree models are again used to pick out features of the derivational history that are predictive for a certain parsing decision. SPATTER began the trend of decomposing local phrase structure trees into individual parsing decisions, but rather than using a variant of dependency grammar, as in most other work, it used a somewhat odd technique of predicting which way the branch above a node pointed.

In SPATTER, a parse tree is encoded in terms of *words*, part of speech *tags*, non-terminal *labels*, and *extensions*, which encode the tree shape. Tagging was done as part of the parsing process. Since the grammar is fully lexicalized, the word and tag of the head child is always carried up to non-terminal nodes. If we start with some words and want to predict the subtree they form, things look something like this:

(12.32)



A node predicts an extension which expresses the type of the line above it connecting it to the parent node. There are five extensions: for subtrees with two or more branches, **right** is assigned to the leftmost child, **left** is assigned to the rightmost child, and **up** is assigned to any children in between, while **unary** is assigned to an 'only child' and **root** is assigned to the root node of the tree. (Note that right and left are thus switched!)

These features, including the POS tags of the words, are predicted by decision-tree models. For one node, features are predicted in terms of features of surrounding and lower nodes, where these features have already been determined. The models use the following questions (where X is one of the four features mentioned above):

- What is the *X* at the {current node/node {1/2} to the {left/right}}?
- What is the *X* at the current node's {first/second} {left/right}-most child?
- How many children does the node have?
- What is the span of the node in words?
- [For tags:] What are the two previous POS tags?

The parser was allowed to explore different derivation sequences, so it could start working where the best predictive information was available (although in practice possible derivational orders were greatly constrained). The probability of a parse was found by summing over derivations.

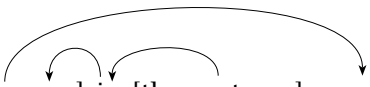
Some features of SPATTER, such as the extensions feature, were rather weird, and overall the result was a large and complex system that required a great deal of computer power to train and run (the decision tree training and smoothing algorithms were particularly computationally intensive). But there was no doubting its success. SPATTER showed that one could automatically induce from treebank data a successful statistical parser which clearly outperformed any existing hand-crafted parser in its ability to handle naturally occurring text.

12.2.3 Dependency-based models

Collins (1996)

More recently Collins (1996; 1997) has produced probabilistic parsing models from treebank data that are simpler, more intuitive, and more quickly computable than those explored in the preceding subsection, but which perform as well or better.

Collins (1996) introduces a lexicalized generally Dependency Grammar-like framework, except that baseNP units in the Penn Treebank are treated as chunks (using chunks in parsing in this way is reminiscent of the approach of Abney (1991)). The original model was again a parsing model. A sentence was represented as a bag of its baseNPs and other words (*B*) with dependencies (*D*) between them:

(12.33) 

Then:

$$P(t|s) = P(B, D|s) = P(B|s) \times P(D|s, B)$$

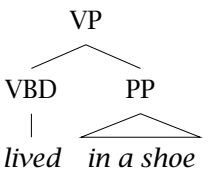
Tagging was an independent process, and was performed by the maximum entropy tagger of Ratnaparkhi (1996). The probability estimate for baseNPs uses the idea of Church (1988) for identifying NPs (see section 10.6.2). Each gap G_i between words is classified as either the start or end of an NP, between two NPs or none of the above. Then the probability of a baseNP β of length m starting at w_u is given in terms of the predicted gap features as:

$$P(\beta|s) = \prod_{i=u+1}^{u+m} \hat{P}(G_i|w_{i-1}, t_{i-1}, w_i, t_i, c_i)$$

where c_i represents whether there is a 'comma' between the words or not. Deleted interpolation is used to smooth this probability.

For the dependency model, Collins replaced each baseNP with its head word and removed punctuation to give a reduced sentence. But punctuation is used to guide parsing. Part of what is clever about Collins' approach is that he works directly with the phrase structures of the Penn Treebank, but derives a notation for dependencies automatically from them. Dependencies are named based on the head and two child constituents. So if one has a subtree as in (12.34), the dependency between the PP and verb is labeled VBD_VP_PP.

(12.34)



In other words, the dependency names are derived from purely categorial labels, but end up capturing much of the functional information that one would like to use in a parser. Nevertheless, the system does still have a few limitations – for instance, these dependency labels do not capture the difference between the two objects of a ditransitive verb.

Each dependency is assumed to be independent – a somewhat unrealistic assumption. Then, each word w_m apart from the main predicate of the sentence will be dependent on some head h_{w_m} via a dependency

relationship $R_{w_m,h_{w_m}}$. Thus D can be written as a set of dependencies $\{d(w_i, h_{w_i}, R_{w_i,h_{w_i}})\}$, and we have that:

$$P(D|S,B) = \prod_{j=1}^{n} P\left(d(w_j, h_{w_j}, R_{w_j,h_{w_j}})\right)$$

Collins calculates the probability that two word-tag pairs $\langle w_i, t_i\rangle$ and $\langle w_j, t_j\rangle$ appear in the same reduced sentence with relationship R within the Penn treebank in the obvious way. He counts up how common one relationship is compared to the space of all relationships:

$$\hat{F}(R|\langle w_i, t_i\rangle, \langle w_j, t_j\rangle) = \frac{C(R, \langle w_i, t_i\rangle, \langle w_j, t_j\rangle)}{C(\langle w_i, t_i\rangle, \langle w_j, t_j\rangle)}$$

And then he normalizes this quantity to give a probability.

This model was then complicated by adding conditioning based on the 'distance' over which the dependency stretched, where distance was evaluated by an ad hoc function that included not only the distance, but direction, whether there was an intervening verb, and how many intervening commas there were.

The parser used a beam search with various pruning heuristics for efficiency. The whole system can be trained in 15 minutes, and runs quickly, performing well even when a quite small beam is used. Collins' parser slightly outperforms SPATTER, but the main advance seems to be in building a much simpler and faster system that performs basically as well. Collins also evaluates his system both using and not using lexical information and suggests that lexical information gives about a 10% improvement on labeled precision and recall. The odd thing about this result is that the unlexicalized version ends up performing rather worse than Charniak's PCFG parser. One might hypothesize that while splitting up local subtrees into independent dependencies is useful for avoiding data sparseness when dealing with a lexicalized model, it nevertheless means that one doesn't capture some (statistical) dependencies which are being profitably used in the basic PCFG model.

A lexicalized dependency-based language model

Collins (1997) redevelops the work of Collins (1996) as a generative language model (whereas the original had been a probabilistically deficient parsing model). He builds a sequence of progressively more complex

models, at each stage getting somewhat improved performance. The general approach of the language model is to start with a parent node and a head and then to model the successive generation of dependents on both the left and right side of the head. In the first model, the probability of each dependent is basically independent of other dependents (it depends on the parent and head nodes' category, the head lexical item, and a final composite feature that is a function of distance, intervening words, and punctuation). Dependents continue to be generated until a special pseudo-nonterminal STOP is generated.

Collins then tries to build more complex models that do capture some of the (statistical) dependencies between different dependents of a head. What is of particular interest is that the models start bringing in a lot of traditional linguistics. The second model makes use of the argument/adjunct distinction and models the subcategorization frames of heads. A subcategorization frame is predicted for each head, and the generation of dependents is additionally conditioned on the bag of subcategorized arguments predicted that have not yet been generated. A problem caused by trying to model subcategorization is that various subcategorized arguments may not be overtly present in their normal place, due to processes like the implicit object alternation (section 8.4) or *Wh*-movement. In the final model Collins attempts to incorporate *Wh*-movement into the probabilistic model, through the use of traces and coindexed fillers (which are present in the Penn treebank). While the second model performs considerably better than the first, this final complication is not shown to give significantly better performance.

12.2.4 Discussion

Some overall parsing performance figures for some roughly comparable systems are shown in table 12.6.[17] At the time of writing, Collins' results are the best for a broad coverage statistical parser. It remains an open research problem to see whether one can weld useful elements of the IBM work (such as using held out data to estimate model parameters, the use

17. All these systems were trained on the Penn Treebank, and tested on an unseen test set of sentences of 2–40 words, also from the Penn Treebank. However, various details of the treatment of punctuation, choosing to ignore certain non-terminal distinctions, etc., nevertheless mean that the results are usually not *exactly* comparable. The results for SPATTER are the results Collins (1996) gives for running SPATTER on the same test set as his own parsers, and differ slightly from the results reported in (Magerman 1995).

	Sentences of ≤ 40 words			
	% LR	% LP	CB	% 0 CBs
Charniak (1996) PCFG	80.4	78.8	n/a	n/a
Magerman (1995) SPATTER	84.6	84.9	1.26	56.6
Collins (1996) best	85.8	86.3	1.14	59.9
Charniak (1997a) best	87.5	87.4	1.00	62.1
Collins (1997) best	88.1	88.6	0.91	66.5

Table 12.6 Comparison of some statistical parsing systems. LR = labeled recall, LP = labeled precision, CB = crossing brackets, n/a means that a result is not given (Charniak (1996) gives a result of 87.7% for non-crossing accuracy).

of decision trees, and more sophisticated deleted estimation techniques) with the key ideas of Collins' work to produce even better parsers. Additionally, we note that there are several other systems with almost as good performance, which use quite different parsing techniques, and so there still seems plenty of room for further investigation of other techniques. For instance, Charniak (1997a) uses probability estimates for conventional grammar rules (suitably lexicalized). The rule by which to expand a node is predicted based on the the node's category, its parent's category, and its lexical head. The head of each child is then predicted based on the child's category and the parent node's category and lexical head. Charniak provides a particularly insightful analysis of the differences in the conditioning used in several recent state-of-the-art statistical parsers and of what are probably the main determinants of better and worse performance.

Just as in tagging, the availability of rich lexical resources (principally, the Penn treebank) and the use of statistical techniques brought new levels of parsing performance. However, we note that recent incremental progress, while significant, has been reasonably modest. As Charniak (1997a: 601) points out:

> This seems to suggest that if our goal is to get, say, 95% average labeled precision and recall, further incremental improvements on this basic scheme may not get us there.

Qualitative breakthroughs may well require semantically richer lexical resources and probabilistic models.

12.3 Further Reading

A variety of work on grammar induction can be found in the biennial proceedings of the International Colloquium on Grammar Inference (Carrasco and Oncina 1994; Miclet and de la Higuera 1996; Honavar and Slutzki 1998).

The current generation of work on probabilistic parsing of unrestricted text emerged within the DARPA Speech and Natural Language community. Commonly cited early papers include (Chitrao and Grishman 1990) and (Magerman and Marcus 1991). In particular, Magerman and Marcus make early reference to the varying structural properties of NPs in different positions.

Another thread of early work on statistical parsing occurred at the University of Lancaster. Atwell (1987) and Garside and Leech (1987) describe a constituent boundary finder that is similar to the NP finding of (Church 1988). A PCFG trained on a small treebank is then used to choose between possible constituents. Some discussion of the possibilities of using simulated annealing also appears. They suggest that their system could find an "acceptable" parse about 50% of the time.

Another important arena of work on statistical parsing is work within the pattern recognition community, an area pioneered by King-Sun Fu. See in particular (Fu 1974).

An approachable introduction to statistical parsing including part-of-speech tagging appears in (Charniak 1997b). The design of the Penn Treebank is discussed in (Marcus et al. 1993) and (Marcus et al. 1994). It is available from the Linguistic Data Consortium.

PARSEVAL MEASURES

The original *PARSEVAL measures* can be found in (Black et al. 1991) or (Harrison et al. 1991). A study of various parsing evaluation metrics, their relationships, and appropriate parsing algorithms for different objective functions can be found in (Goodman 1996).

DEPENDENCY GRAMMAR

The ideas of *dependency grammar* stretch back into the work of medieval Arab grammarians, but received a clear formal statement in the work of Tesnière (1959). Perhaps the earliest work on probabilistic dependency grammars was the Probabilistic Link Grammar model of Lafferty et al. (1992). Except for one particular quirky property where a word can be bi-linked in both directions, link grammar can be thought of as a notational variant of dependency grammar. Other work on dependency-based statistical parsers includes Carroll and Charniak (1992).

We have discussed only a few papers from the current flurry of work

on statistical parsing. Systems with very similar performance to (Collins 1997), but very different grammar models, are presented by Charniak (1997a) and Ratnaparkhi (1997a). See also (Eisner 1996) for another recent approach to dependency-based statistical parsers, quite similar to (Collins 1997).

Most of the work here represents probabilistic parsing with a context-free base. There has been some work on probabilistic versions of more powerful grammatical frameworks. Probabilistic TAGs (*Tree-Adjoining Grammars*) are discussed by Resnik (1992) and Schabes (1992). Early work on probabilistic versions of unification grammars like Head-driven Phrase Structure Grammar (Pollard and Sag 1994) and Lexical-Functional Grammar (Kaplan and Bresnan 1982), such as (Brew 1995), used improper distributions, because the dependencies within the unification grammar were not properly accounted for. A firmer footing for such work is provided in the work of Abney (1997). See also (Smith and Cleary 1997). Bod et al. (1996) and Bod and Kaplan (1998) explore a DOP approach to LFG.

TREE-ADJOINING GRAMMARS

Transformation-based learning has also been applied to parsing and grammar induction (Brill 1993a,c; Brill and Resnik 1994). See chapter 10 for a general introduction to the transformation-based learning approach.

TRANSFORMATION-BASED LEARNING

Hermjakob and Mooney (1997) apply a non-probabilistic parser based on machine learning techniques (decision lists) to the problem of treebank parsing, and achieve quite good results. The main take-home message for future Statistical NLP research in their work is the value they get from features for semantic classes, whereas most existing Statistical NLP work has tended to overemphasize syntactic features (for the obvious reason that they are what is most easily obtained from the currently available treebanks).

Chelba and Jelinek (1998) provide the first clear demonstration of a probabilistic parser outperforming a trigram model as a language model for *speech recognition*. They use a lexicalized binarized grammar (essentially equivalent to a dependency grammar) and predict words based on the two previous heads not yet contained in a bigger constituent.

SPEECH RECOGNITION

Most of the exposition in this chapter has treated parsing as an end in itself. Partly because parsers do not perform well enough yet, parsing has rarely been applied to higher-level tasks like speech recognition and language understanding. However, there is growing interest in *semantic parsing*, an approach that attempts to build a meaning representation of a sentence from its syntactic parse in a process that integrates syntactic and semantic processing. See (Ng and Zelle 1997) for a recent overview

SEMANTIC PARSING

article. A system that is statistically trained to process sentences all the way from words to discourse representations for an airline reservation application is described by Miller et al. (1996).

12.4 Exercises

Exercise 12.1 [⋆]

The second sentence in the *Wall Street Journal* article referred to at the start of the chapter is:

(12.35) The agency sees widespread use of the codes as a way of handling the rapidly growing mail volume and controlling labor costs.

Find at least five well-formed syntactic structures for this sentence. If you cannot do this exercise, you should proceed to exercise 12.2.

Exercise 12.2 [⋆⋆]

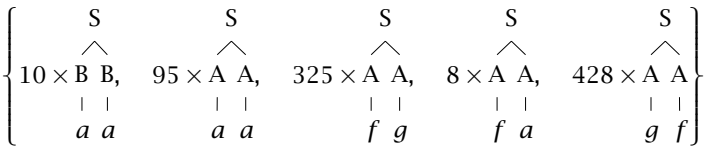
Write a context-free grammar parser, which takes a grammar of rewrite rules, and uses it to find all the parses of a sentence. Use this parser and the grammar in (12.36) to parse the sentence in exercise 12.1. (The format of this grammar is verbose and ugly because it does not use the abbreviatory conventions, such as optionality, commonly used for phrase structure grammars. On the other hand, it is particularly easy to write a parser that handles grammars in this form.) How many parses do you get? (The answer you should get is 83.)

(12.36)
a. S → NP VP
b. VP → { VBZ NP | VBZ NP PP | VBZ NP PP PP }
c. VPG → VBG NP
d. NP → { NP CC NP | DT NBAR | NBAR }
e. NBAR → { AP NBAR | NBAR PP | VPG | N | N N }
f. PP → P NP
g. AP → { A | RB A }
h. N → { *agency*, *use*, *codes*, *way*, *mail*, *volume*, *labor*, *costs*}
i. DT → { *the*, *a* }
j. VBZ → *sees*
k. A → { *widespread*, *growing* }
l. P → { *of*, *as* }
m. VBG → { *handling*, *controlling* }
n. RB → *rapidly*
o. CC → *and*

While writing the parser, leave provision for attaching probabilities to rules, so that you will be able to use the parser for experiments of the sort discussed later in the chapter.

Exercise 12.3 [⋆]

In chapter 11, we suggested that PCFGs have a bad bias towards using nonterminals with few expansions. Suppose that one has as a training corpus the treebank given below, where '$n\times$' indicates how many times a certain tree appears in the training corpus. What PCFG would one get from the treebank (using MLE as discussed in the text)? What is the most likely parse of the string '*a a*' using that grammar. Is this a reasonable result? Was the problem of bias stated correctly in chapter 11? Discuss.



Exercise 12.4 [⋆⋆]

Can one combine a leftmost derivation of a CFG with an n-gram model to produce a probabilistically sound language model that uses phrase structure? If so, what kinds of independence assumptions does one have to make? (If the approach you work out seems interesting, you could try implementing it!)

Exercise 12.5 [⋆]

While a PLCG can have different probabilities for a certain expansion of NP in subject position and object position, we noted in a footnote that a PLCG could not capture the different distributions of NPs as first and second objects of a verb that were shown in table 12.4. Explain why this is so.

Exercise 12.6 [⋆⋆⋆]

As shown by Demers (1977), left-corner parsers, top-down parsers and bottom-up parsers can all be fit within a large family of Generalized Left-Corner Parsers whose behavior depends on how much of the input they have looked at before undertaking various actions. This suggests other possible probabilistic models implementing other points in this space. Are there other particularly useful points in this space? What are appropriate probabilistic models for them?

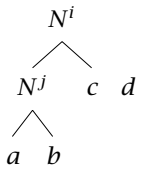
Exercise 12.7 [⋆⋆]

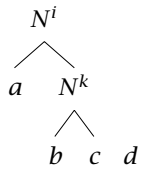
In section 12.2.1 we pointed out that a non-lexicalized parser will always choose the same attachment in the same structural configuration. However, thinking about the issue of PP attachment discussed in section 8.3, that does not quite mean that it must always choose noun attachments or always choose verb attachments for PPs. Why not? Investigate in a corpus whether there is any utility in being able to distinguish the cases that a PCFG can distinguish.

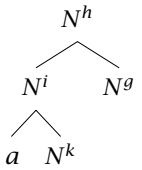
Exercise 12.8 [⋆]

The aim of this exercise is to appreciate why one cannot build a Viterbi algorithm for DOP parsing. For PCFG parsing, if we have built the two constituents/partial derivations shown in (12.37a) and (12.37b), and $P(N^i)$ in (12.37a) $> P(N^i)$ in (12.37b), then we can discard (12.37b) because any bigger tree built using (12.37b) will have a lower probability than a tree which is otherwise identical but substitutes (12.37a). But we cannot do this in the DOP model. Why not? Hint: Suppose that the tree fragment (12.37c) is in the corpus.

(12.37)

a. 

b. 

c. 

Exercise 12.9 [⋆⋆⋆]

Build, train, and test your own statistical parser using the Penn treebank. Your results are more likely to be useful to others if you chose some clear hypothesis to explore.