# Function Tagging

by
Don Blaheta
A. M., Brown University, 2003
Sc. M., Brown University, 1999
B. S., Quincy University, 1997

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

Providence, Rhode Island
May 2004

© Copyright 2004 by Don Blaheta

Abstract of "Function Tagging" by Don Blaheta, Ph.D., Brown University, May 2004.

Function tags are a context-sensitive annotation applied to words and phrases of natural language text, marking their syntactic or semantic role within a larger utterance. As researchers improve results on various other problems in "pure" natural language processing (e.g part-of-speech tagging, parsing), those who work in the more "applied" NLP fields (e.g. question-answering, temporal analysis) are seeking more powerful sorts of linguistic annotation as input for their own systems. Hence, function tags.

In the first part of the thesis, I present the problem of function tagging: why it is an interesting problem, who has worked on similar thing, and what exactly I intend to do. I briefly review the function tags of the Penn treebank, and explain the specific metrics by which I will evaluate my work.

In the second part of the thesis, I introduce the many features that I will use to train a function tagging system, and then I present some systems that make use of them: one using feature trees, one using decision trees (briefly), and one using perceptron models. For each system, I give a brief historical perspective, an overview of where it has been used before and why I think it will be useful in this task. I will then try a number of feature combinations with interesting properties; and finally, present the best-performing tweaked-out version of that system.

Finally, in the third part of the thesis, I bring them all together and discuss the advantages and disadvantages of each system in various situations. More interestingly, I will present an analysis of what features prove to be the most helpful for the different function tagging subtasks. Lastly, I will present a comparison to other systems performing related tasks, and speculate on some interesting future work.

# Vita

**Don Blaheta** was born on 28 March 1978 in Chicago, and continued to reside in Illinois through grade school, high school, and his undergraduate years at Quincy University (in Quincy, Illinois), where he graduated *cum laude* with bachelor's degrees in mathematics and in computer science. He moved to Providence in 1997 to commence studies in the Ph.D. program in Computer Science here at Brown, resulting in an Sc.M. degree in 1999 and culminating in this Ph.D. thesis; he also has taken classes and done research in the Department of Cognitive and Linguistic Sciences, where he received an A.M. degree in Linguistics in 2003. This Fall he will be returning to Illinois, where he has been appointed to the position of Assistant Professor of Computer Science at Knox College, in Galesburg.

# Acknowledgements

I would first of all like to thank my advisor, Eugene Charniak, for giving me just the right level of prodding over the years to keep me going; and for being patient with my work across the street in linguistics (especially in the late stages of the linguistics thesis, when I was getting virtually no work done on this one!). Mark Johnson was a constant resource—both on my computer science research and on my linguistics research—and Michael Collins gave a lot of extremely helpful technical advice that helped me to actually understand the vagaries of the perceptron algorithm.

This is an excellent place to thank the National Science Foundation for funding me, through the Integrative Graduate Education and Research Traineeship (IGERT) program—a great deal that enabled me to broaden my horizons, and gave me the time to pursue this master's degree outside my primary field.

And now, before the music comes up and the microphone descends into the floor, I'd like to thank the members of the Brown Ballroom Dance Team, the Brown Renaissance Singers, the Brown-RISD Catholic Community, and the `imsasun` regulars, as well as the whole Coconut Lounge crowd, for their assistance in sanity maintenance. Special thanks go to Matt Sherman, for taking time off from his own thesis for some excellent and much-needed late-night conversation; to Theresa Ross, for great conversations, a fabulous roadtrip, and my newfound obsession with knitting; to Kathleen Corriveau, for many years of fun dancing and good times; to Sharon Goldwater, for sharing my interests in computer science, linguistics, *and* dancing, and for her insight into and perspective on all of those things; and to Mike Kimmitt, for his contagious political fervour, for his welcoming spirit, and most for his dozen-years-strong friendship.

Last but never least, I need to thank my family, and especially my mom, without whom none of this would have been possible.

Don Blaheta
*Providence, RI*
*August 2003*

# Contents

**List of Tables** xi

**List of Figures** xiii

**1 Introduction** 1

**2 Function tags** 3

2.1 The Penn Treebank II function tag set . . . . . 4

2.1.1 The curious case of the CLR tag . . . . . 5

2.2 Other types of function tags . . . . . 6

2.2.1 Collins (1997): complement marking . . . . . 6

2.2.2 NEGRA (1997): 'grammatical function tags' . . . . . 6

2.2.3 Carroll, Briscoe, and Sanfilippo (1998): 'grammatical relations' . . . . . 7

2.2.4 FrameNet (1998): 'frame elements' . . . . . 8

2.2.5 Böhmová, Hajič, Hajičová, and Hladká (2000): 'functors' . . . . . 8

2.3 Uses for function tags . . . . . 9

2.4 Evaluating performance on the task . . . . . 10

**3 Features** 13

3.1 Binary vs. multivalued features . . . . . 13

3.2 Some possibly useful features . . . . . 14

3.2.1 Label . . . . . 14

3.2.2 cc-Label . . . . . 14

3.2.3 Head . . . . . 14

3.2.4 Head part-of-speech (POS) . . . . . 16

3.2.5 Alt head . . . . . 16

3.2.6 Alt head POS . . . . . 16

3.2.7 Function tags . . . . . 16

3.2.8 Label clusters . . . . . 16

3.2.9 Word clusters . . . . . 17

3.3 Making use of relatives . . . . . 17

**4 Feature trees** **19**
4.1 A brief history and overview of maximum entropy . . . . 19
4.2 From interpolation to feature trees . . . . 21
4.3 Experimental design . . . . 24
4.4 Results . . . . 24
4.5 Incorporating function tagging into parsing . . . . 27

**5 Decision trees** **31**
5.1 Decision trees in function tagging . . . . 31
5.2 Why we abandoned decision trees . . . . 32

**6 Perceptrons** **33**
6.1 A high level view . . . . 33
6.2 Multi-valued classification . . . . 34
6.2.1 The algorithm . . . . 36
6.3 Voted perceptrons . . . . 37
6.3.1 Sparse voted perceptron . . . . 39
6.4 Kernel-based perceptrons . . . . 40
6.5 Feature sets . . . . 42
6.6 Results . . . . 42

**7 Analysis** **49**
7.1 Feature trees vs. perceptrons . . . . 49
7.2 Helpful features . . . . 50
7.3 Related work . . . . 51
7.3.1 Collins (1997) . . . . 51
7.3.2 Brants, Skut, and Krenn (1997) . . . . 51
7.3.3 Preiss (2003) . . . . 53
7.3.4 Gildea and Jurafsky (2002) . . . . 54
7.4 Error analysis and fixing the treebank . . . . 55
7.5 Parse error . . . . 56
7.6 Treebank error . . . . 57
7.6.1 Type A: Detectable errors . . . . 57
7.6.2 Type B: Fixable errors . . . . 58
7.6.3 Type C: Systematic inconsistency . . . . 59
7.6.4 `tsed` . . . . 60
7.6.5 Training data . . . . 61
7.6.6 Testing data . . . . 62
7.6.7 Ethical considerations . . . . 62
7.6.8 Practical considerations . . . . 63

7.6.9 Experimental results . . . 64
7.6.10 Further notes on error correction . . . 65

**8 Conclusion and future work** **67**
8.1 Contributions . . . 67
8.2 Future work . . . 68

**A Function tag descriptions** **69**
A.1 Syntactic tags . . . 69
A.1.1 DTV—Dative . . . 69
A.1.2 LGS—Logical subject . . . 69
A.1.3 PRD—Predicative . . . 70
A.1.4 PUT—Locative complement of 'put' . . . 70
A.1.5 SBJ—Subject . . . 70
A.1.6 VOC—Vocative . . . 70
A.2 Semantic tags . . . 70
A.2.1 ADV—Adverbial . . . 71
A.2.2 BNF—Benefactive . . . 71
A.2.3 DIR—Direction . . . 71
A.2.4 EXT—Extent . . . 71
A.2.5 LOC—Locative . . . 71
A.2.6 MNR—Manner . . . 71
A.2.7 NOM—Nominative . . . 72
A.2.8 PRP—Purpose . . . 72
A.2.9 TMP—Temporal . . . 72
A.3 Topicalisation (TPC) . . . 72
A.4 Miscellaneous . . . 72
A.4.1 CLF—'It'-Cleft . . . 72
A.4.2 CLR—"Closely related" . . . 73
A.4.3 HLN—Headline . . . 73
A.4.4 TTL—Title . . . 73

**B `tsed`** **75**
B.1 The command-line . . . 75
B.1.1 General options . . . 76
B.1.2 `tgrep` options . . . 76
B.1.3 `tsed` options . . . 76
B.1.4 `wsjsed` options . . . 77
B.2 Search patterns . . . 77
B.2.1 Search pattern operators . . . 78

B.2.2 Search pattern negation: ! . . . 80
B.2.3 Alternate operator characters . . . 81
B.2.4 Search pattern examples . . . 81
B.3 Replacement patterns . . . 82
B.3.1 Referring to the matched spattern . . . 82
B.3.2 The rpattern types . . . 82
B.3.3 Replacing subpatterns . . . 83
B.3.4 Some notes on the inner workings . . . 84
B.3.5 Example rpatterns . . . 84
B.4 Modifying the Penn treebank . . . 85
B.4.1 Comments . . . 86
B.4.2 One-line commands . . . 86
B.4.3 Batch commands . . . 86
B.5 Installation . . . 86
B.5.1 Executables . . . 87
B.5.2 Source . . . 87

**C Treebank corrections** **89**

**D The Principle of Indifference** **99**

# List of Tables


2.1 A comparison between the NEGRA tagset and Penn's . . . . . . . . . . . . . . . . . . 7

4.1 Performance of tree in Figure 4.3 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
4.2 F-measure from various test runs on Section 24 . . . . . . . . . . . . . . . . . . . . . 26
4.3 Key to numeric feature IDs in Figure 4.2 on page 26 . . . . . . . . . . . . . . . . . . 27
4.4 Performance of new trees on Section 23 . . . . . . . . . . . . . . . . . . . . . . . . . 28

6.1 Abbreviations used in feature set names . . . . . . . . . . . . . . . . . . . . . . . . 43
6.2 Features used in each feature set . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
6.3 Results of the basic perceptron and sparse voted perceptron, after 20 epochs of training, tested on section 24 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
6.4 Voted perceptron results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
6.5 Kernel perceptron results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
6.6 SVP results with fullcorp2-ccp training, broken down by tag . . . . . . . . . . . . . . 48

7.1 A comparison between our work and Brants et al . . . . . . . . . . . . . . . . . . . . 53
7.2 Comparison against Preiss hand-built tagger . . . . . . . . . . . . . . . . . . . . . . 54
7.3 Some results from Gildea and Jurafsky (2002) . . . . . . . . . . . . . . . . . . . . . 55
7.4 Analysis of reported errors . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
7.5 Function tagging results, adjusted for treebank error . . . . . . . . . . . . . . . . . . 64

B.1 Operation substitution characters . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81

# List of Figures


2.1 Penn treebank function tags . . . . . 4
2.2 Categories of function tags and their relative frequencies . . . . . 5
2.3 Measures of accuracy . . . . . 10

3.1 Head-finding algorithm . . . . . 15
3.2 Which constituent types make good heads vs. tolerable heads . . . . . 15
3.3 Label clusters . . . . . 17

4.1 A feature chain . . . . . 23
4.2 A feature tree . . . . . 23
4.3 The generic feature tree used to guess function tags in the original paper . . . . . 24
4.4 The feature tree used to guess syntactic tags . . . . . 27
4.5 The feature tree used to guess semantic tags . . . . . 27
4.6 An example illustrating why integrating systems might help . . . . . 28
4.7 An example illustrating why integrating systems doesn't help . . . . . 29
4.8 An example illustrating why integrating systems might still help . . . . . 29

5.1 Preliminary decision tree results . . . . . 32

6.1 A linearly-separable dataset, and some valid separators . . . . . 34
6.2 A non-linearly-separable dataset . . . . . 35
6.3 A tri-valued dataset . . . . . 36
6.4 The perceptron algorithm: training . . . . . 36
6.5 The perceptron algorithm: testing . . . . . 37
6.6 The voted perceptron algorithm: training . . . . . 38
6.7 The voted perceptron algorithm: testing . . . . . 38
6.8 The dual form of the perceptron algorithm: training . . . . . 39
6.9 The dual form of the perceptron algorithm: testing . . . . . 39
6.10 The surface $z = xy$ . . . . . 40
6.11 A plane in 3D corresponds to a hyperbola in 2D . . . . . 41
6.12 Basic perceptron using ecbasic feature set for semantic tags . . . . . 43

6.13 Performance of the basic perceptron on the semantic tags, given similar feature sets 45

7.1 A sample sentence with NEGRA-style annotation . . . . . . . . . . . . . . . . . . . . 52
7.2 SBAR and conditioning info . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
7.3 SBAR and conditioning info, as parsed . . . . . . . . . . . . . . . . . . . . . . . . . 57
7.4 A function tag error of Type A . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
7.5 A part-of-speech error of Type $B_1$ . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
7.6 A parse error of Type $B_2$ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59

# Chapter 1

# Introduction

As the state of the art in Natural Language Processing advances, we have seen progressively higher levels of annotation made available. On top of part-of-speech tags, which we've been fairly good at for a while now, we have seen parse structure and phrase labels; the next natural step is to mark the role each of these phrases plays in the overall utterance—with function tags.

Like the other forms of annotation, function tagging is not to be seen as an end in and of itself. Knowing that a given phrase is, say, the topic of a sentence, or a locative modifier, is not likely to be useful or interesting to the typical end user. However, this information is important as a tool for other natural language applications. A dialogue system could certainly make use of information about marked topics. Question answering systems can make use of semantic tags to note that "where" questions will be answered by locative or directional phrases, "when" by temporal, and so on. Machine translation systems can make use of nearly all of the function tags to refine alignments (as the subject of a sentence will nearly always align with the subject of a parallel sentence in another language) as well as for generation (as the case of a noun phrase will often have a direct relationship to its syntactic or semantic function tag: subect? location? manner/instrument?)

Researchers at the University of Pennsylvania have given us the Penn Treebank, and beginning with the second edition of that corpus, they have included function tag information therein. This thesis will investigate some methods for using this corpus—already heavily mined for parsing applications—to train a system to mark parsed text with the Penn function tags.

Inevitably, some portions of the task are easier than others. In easier cases, a function tagger can determine its prediction based just on the local structure of the parse, with just one or two phrase labels—"at a glance", one might say. The tricky part of function tagging comes when distinguishing phrases of the same structure that differ by just one word—for instance, telling 'in Budapest', which is locative, from 'in April', which is temporal. There are many cases where a single preposition can give rise to two or three different function tags; the word 'by' is associated with *five*. In the following sentences, only the objects of the preposition differ:

The volume was turned up by John (LGS)

by 30 dB (EXT)
by the DJ table (LOC)
by a twist of the knob (MNR)
by 11pm (TMP)

That object is even always a noun phrase, and yet the five cases receive five different tags. As we shall see, it is cases like this—where lexical information is needed to distinguish possible tags—that will be hardest for a function tagging system to process. Nevertheless, the problem is not intractable. In the ensuing chapters, we will show that certain types of system are capable of attaining relatively high accuracy—over 98% on syntactic tags and 80% on the (harder) semantic tags.

# Chapter 2

# Function tags

We think it will be useful to begin by formally defining the term that titles the thesis:

> A *function tag* is an annotation, chosen from a relatively small, discrete set of possible annotations, that is placed on a phrase to indicate that phrase's relationship to the rest of the utterance that contains it.

Note that this definition is not meant to include the basic syntactic category of a phrase, such as "noun phrase" or "subordinate clause"; these terms (which we will refer to as *labels*) indicate where a given batch of words might be able to go. A function tag indicates specifically how a given batch of words *in context* relates to its neighbours and the rest of the sentence. While a syntactic label can often be assigned with some confidence to a phrase out of context, a function tag usually cannot. One can look at the phrase "the stock market" and immediately label it a noun phrase, but its function might well be as a subject:

> *The stock market* closed early today.

Or a direct object:

> Investment bankers monitor *the stock market* carefully.

Or as the agent in a passive construction:

> Alex was always fascinated by *the stock market*.

Or perhaps as something else entirely.

Function tags may be basically syntactic in nature (as with those in the previous paragraph), but they may also have a more semantic component. When annotating adjuncts, in particular, it makes sense to mark what type of modification is being performed, e.g. time, location, or manner. Other function tags are surely possible (pragmatics, discourse structure, etc.), but in this work we focus primarily on syntactic and semantic function tags, as these comprise the majority of the function tags in the Penn Treebank, the corpus with which we will be working.

| | | | |
|---|---|---|---|
| ADV | Non-specific adverbial | MNR | Manner |
| BNF | Benefactive | NOM | Nominal |
| CLF | It-cleft | PRD | Predicate |
| CLR | 'Closely related' | PRP | Purpose |
| DIR | Direction | PUT | Locative complement of 'put' |
| DTV | Dative | SBJ | Subject |
| EXT | Extent | TMP | Temporal |
| HLN | Headline | TPC | Topic |
| LGS | Logical subject | TTL | Title |
| LOC | Location | VOC | Vocative |

Figure 2.1: Penn treebank function tags

## 2.1 The Penn Treebank II function tag set

The Penn Treebank II (Marcus et al., 1995) is a corpus of hand-annotated text that includes part-of-speech tags, full sentence parses (including empty nodes and movement annotation), and function tags.[1] We have used the Wall Street Journal portion of that corpus, which is approximately one million words of text from that publication (from 1989), and for the remainder of this work will use the word "treebank" to refer to this WSJ portion of the Penn Treebank II, except where noted.

The Treebank II bracketing guidelines identify twenty function tags for annotators to use, as listed in Figure 2.1. A brief description of each can be found in Appendix A; more complete documentation can be found in the corpus annotation guidelines (Bies et al., 1995).

Careful consideration of the guidelines and the corpus itself suggested a fairly natural division of the function tags into four sets of mutually exclusive tags, shown in Figure 2.2 on the facing page. They roughly correspond to categories set out in the guidelines, with some variations. The "syntactic" category is largely untouched, except that Topicalisation (TPC) was removed; that tag can co-occur with various other syntactic tags. The so-called "form/function discrepancy" category, which included the Adverbial (ADV) and Nominal (NOM) tags, was merged in with the "adverbial" category to form our "semantic" category: though ADV and NOM share some properties with the syntactic tags, NOM does sometimes co-occur with some of the syntactic tags; hence it cannot share a category with them, due to the mutual exclusivity criterion. Thus, the four function tag categories we will be using throughout this work: Syntactic, Semantic, Topicalisation, and Miscellaneous.[2] Any given constituent will have at most one function tag per category.[3] In actual practice, constituents with tags from all four categories do not occur, although some have tags from three categories (rarely).

Also in Figure 2.2 are the number of times each tag occurs in Section 24 of the treebank (the

[1] It is available from the Linguistic Data Consortium as Catalogue No. LDC95T7; see `http://www.ldc.upenn.edu/Catalog/` for more details.

[2] In earlier publications we used the terms 'Grammatical' and 'Form/function' for Syntactic and Semantic, respectively.

[3] There is a single exception in the treebank: one constituent is tagged -LOC-MNR, but this appears to be an error.

| | Occurrences in §24 | Within category | Total |
|---|---|---|---|
| **All nonterminals** | 27075 | | |
| **Syntactic** | 3066 | 100.00% | 11.32% |
| DTV | 15 | 0.49% | 0.06% |
| LGS | 98 | 3.20% | 0.36% |
| PRD | 606 | 19.77% | 2.24% |
| PUT | 11 | 0.36% | 0.04% |
| SBJ | 2333 | 76.09% | 8.62% |
| VOC | 3 | 0.10% | 0.01% |
| **Semantic** | 2050 | 100.00% | 7.57% |
| ADV | 206 | 10.05% | 0.76% |
| BNF | 1 | 0.05% | 0.00% |
| DIR | 123 | 6.00% | 0.45% |
| EXT | 52 | 2.54% | 0.19% |
| LOC | 584 | 28.49% | 2.16% |
| MNR | 110 | 5.37% | 0.41% |
| NOM | 131 | 6.39% | 0.48% |
| PRP | 105 | 5.12% | 0.39% |
| TMP | 738 | 36.00% | 2.73% |
| **Topicalisation (TPC)** | 119 | 100.00% | 0.44% |
| **Miscellaneous** | 35 | 100.00% | 0.13% |
| CLF | 1 | 2.86% | 0.00% |
| HLN | 11 | 31.43% | 0.04% |
| TTL | 23 | 65.71% | 0.08% |

Figure 2.2: Categories of function tags and their relative frequencies

"development corpus"; see below), along with frequencies. The first percentage on each line represents, of those constituents tagged with *some* tag in this category, how many are tagged with *this* particular tag. The second percentage is out of *all* nonterminal constituents.

### 2.1.1 The curious case of the CLR tag

One of the function tags to be found in the Penn treebank is CLR, which stands for "Closely Related". The bracketing guidelines say that this tag "marks constituents that occupy some middle ground between argument and adjunct of the verb phrase." Their main role seems to be in marking different varieties of phrasal verb, such as 'rely on' or 'put up with'. This is an interesting linguistic phenomenon, and useful to mark.

Unfortunately, it is not exactly a binary phenomenon; the degree to which a verb and some other constituent are phrasal, or "closely related", often lies very much in the eye of the beholder. Even a casual perusal of the development corpus will reveal a few CLR constituents that seem like they shouldn't be, and a large number that aren't but seem like they should be. Every reader, furthermore, will put different constituents in these two sets.

In earlier iterations of this work, we put the CLR tag into the fourth, "miscellaneous" category of function tags. At some 603 such tags in the development corpus, it was by far the dominant member of that category. We recently discovered, however, that its original purpose was to "relieve annotator frustration"[4]: it seems that the annotators saw the phrasal verb phenomenon and wished to mark it. The decision was thus made to create the tag, tell the annotators to "use it however you want"; and then, before compiling the corpus, the distributors were "supposed to strip out the CLR tag." This never happened, of course.

We have thus stopped training, testing, or reporting results on the CLR tag, and restricted the fourth category to its other three, much rarer, members.

## 2.2 Other types of function tags

To our knowledge, there has been no previous attempt at recovering the Penn treebank function tags, as such. There have, however, been a number of projects to annotate some other sorts of function tags.

### 2.2.1 Collins (1997): complement marking

Collins (1997) approaches the problem of distinguishing adjuncts from complements. The motivation here was to add some useful syntactic information and to improve parser performance—by guessing complement status during the parse, the statistics were made a bit cleaner. This paper defines a 'complement' tag, derived from a combination of syntactic label and Penn-style function tag. This boolean condition is then used to train an improved parser.

The system uses a generative model to evaluate parse quality, and the complement information is used as follows. After generating the head-containing child of a constituent (conditioned on the constituent and its head word), left and right subcategorisation frames are chosen conditional on that head-containing child (and the previously-used conditioning information). A subcategorisation frame is simply a bag of labels that are subcategorised by the parent—that is, these labels represent constituents that are complement to the parent constituent. Given the subcategorised frame, then, and all the previous conditioning information, the actual labels of the other children are generated. Collins does not report his results on the complement tagging, but reports that using the complement information improves his parser's accuracy by 0.6%.

### 2.2.2 NEGRA (1997): 'grammatical function tags'

The NEGRA corpus (Uszkoreit and others, 1997) addresses the problem of building a German-language treebank at various levels of automation. Like the Penn WSJ treebank project, NEGRA begins with newspaper text (from the *Frankfurter Rundschau*), which is then POS-tagged, parsed, and function-tagged. The grammar used is a dependency-style grammar, and hence allows for

[4] Mitch Marcus, p.c., 7 July 2002

| | NEGRA | Penn |
|---|---|---|
| Word POS tags | 54 | 36 |
| Punctuation POS tags | 3 | 9 |
| Base phrasal labels | 15 | 27 |
| Coordination phrase labels | 10 | 3[5] |
| Function tags | 45 | 20 |

Table 2.1: A comparison between the NEGRA tagset and Penn's

numerous discontinuous constituents (appropriate for a language like German). Unlike the Penn treebank, this corpus has a function tag on every constituent except the root node, but including the terminals, representing the nature of the dependency between the constituent and its 'parent'. Some of the function tags are direct analogues of Penn function tags, like SB 'subject'; others, such as HD 'head' or OA 'accusative object' are not marked in the Penn treebank. Overall, the two tagsets are comparable, though NEGRA is more fine-grained on words and function tags, while less fine-grained on phrasal labels. See Table 2.1 for a full comparison of the number of tags or labels in each group.

Brants et al. (1997) developed a system to function tag according to this tagset, using Markov models. Their system and their results will be discussed in Section 7.3.2.

### 2.2.3 Carroll, Briscoe, and Sanfilippo (1998): 'grammatical relations'

Carroll et al. (1998) develop a system of "grammatical relations" (GRs) that are intended primarily for use as an evaluation metric. It includes a hierarchy of grammatical function tags (arg subsumes subj, for instance); each of the 20 tags has arguments denoting, typically, what we would call the head, the parent's head, and the alt head of the tagged constituent. Since this system is designed as an evaluation metric for shallow parsing systems, these arguments serve to anchor the function tags to certain parts of the sentence in the absence of a full dependency or phrase-structure tree.

Carroll et al. note specifically that they "are not advocating [the GRs'] use as a parser output representation for use in application tasks," but rather that they are to be "an evaluation method that measures relative correctness of parser analyses with respect to a standard representation that is well-defined and both system- and task-independent." However, this betrays their bias towards wholly or partially hand-built grammars (as, for instance, (Carroll and Briscoe, 2001)). In such a grammar, it is possible to introduce these GRs as part of the parse structure, so that any time a given rule is used, a certain GR is deterministically chosen to go with it.

However, in a parser induced entirely from a training corpus, or even a parser hand-built without the GR evaluation metric in mind, application of the GR tags is itself a task with an imperfect success rate. A perfect example of this can be seen in the parser comparison work in (Preiss, 2003). Preiss uses the GR metric to compare the statistical parsers of Charniak (2000) and Collins (1997) with

[5]These are not in the base Penn tagset, but are used to some advantage both in parsing and function tagging; see Section 3.2.2.

a unification-based parser and a shallow parser. The statistical parsers, not designed to extract these GR tags, suffered in the evaluation: a hand-built GR extractor was run on top of them, and this extractor was not perfect. For example, the performance of Charniak's parser on the ncsubj tag—which we would call SBJ—is given as 82% precision and 70% recall. Since this is the same parser we use in our work, we know that, with the right training, it can do considerably better.

Ultimately, of course, what this means is that the so-called GR *parser*-evaluation metric is really getting at something different than what the statistical parsing community usually means by "parser evaluation"; it is really an evaluation of a function tagger, or rather of the parser+function-tagger combination. It is also not so system-independent as it claims: it is really just another function tag set, and comparing performance of GR-based parser/taggers with performance of Penn Treebank parser/taggers has all the pitfalls of any comparison between multiple tagsets—some tags in each set have no analogues in the other, and some tags in each set correspond to multiple tags in the other. We will return to this problem in Chapter 7, when we compare the performance of various systems.

### 2.2.4 FrameNet (1998): 'frame elements'

Baker et al. (1998) introduce a slightly different sort of function tagging for their FrameNet corpus. Here, the function tags available for tagging a given phrase depend on the predicate frame the phrase is participating in. Rather than a single 'subject' tag, there would be a 'speaker' tag for the 'statement' frame, a 'cognizer' tag for the 'judgement' frame, and so on. The corpus contains about 50,000 sentences hand-tagged with just these function tags (no syntactic structure).

The style of this type of function tag is very different from the others we have mentioned. There are thirteen different *domains*, such as "communication" and "transaction", each of which has numerous *frames*, such as "conversation", "statement", and "judgement". Every frame, then, has its own three to six frame element tags that are shared only by the words within that frame. There are thus many, many more potential tags available in this system, but it is a bounded set; and more importantly, for any given frame the number of different possibilities is small. Thus we do consider this to be another type of function tag.

Gildea and Jurafsky (2002) have constructed a mapping from the frame element tags into a more abstract set of eighteen 'thematic roles', which correspond much more closely to the notion of 'function tags' developed in this work; their results both on the original set and on this reduced set will be discussed in Section 7.3.4.

### 2.2.5 Böhmová, Hajič, Hajičová, and Hladká (2000): 'functors'

In the Prague Dependency Treebank (Böhmová et al., in press), there three layers of annotation that (loosely) correspond to the Penn treebank's part-of-speech tagging, parse structures, and function tags. This highest layer is known as the 'tectogrammatical' layer in the PDT, and is considerably more involved than the Penn treebank's function tags: function words (prepositions, auxiliary verbs)

and punctuation are deleted and sentences are reordered as well. But part of the annotation in this layer is a set of 47 'functors' that indicate the relationship between a dependent and its parent—in other words, function tags.

With 47 of them, of course, the annotation is somewhat finer-grained. Our TMP tag is split into TWHEN, THL, and a few others, to distinguish the "when" modifiers from the "how long" modifiers, the "since when" modifiers, the "until when" modifiers, and so on. Others, such as MANN, map directly back to one of the Penn tags (MNR). In some cases, there is a many-to-many mapping: ACT denotes the 'actor' or 'deep subject' and PAT the 'patient'; which one maps to our SBJ will depend on the sentence (and ACT will sometimes map to our LGS 'logical subject' as well). Ultimately, though, the main thrust is the same, and this function tag set may be the most similar to the Penn tags of all the ones we have reviewed here.

## 2.3 Uses for function tags

This section is still largely speculative; as most of the function tagging work has been fairly recent, researchers who might apply it to other NLP tasks have only just begun to try. Based on a number of personal conversations at recent conferences, however, it seems like many researchers in the field are excited about the prospects. Just a few of the things we (and they) think that function tags might help with:

**Dialogue systems.** Any system that performs high-level sentence understanding, really; the (Preiss, 2003) paper shows that even with an accurate parse, as that output by the Charniak parser, an ad hoc algorithm for extracting even basic syntactic relationships has insufficient coverage. A function-tagged parse should have the information needed to get a good syntactic reading of the sentence. The topicalisation tag in particular seems like it would be handy for dialogue systems, and possibly for anaphora resolution.

**Question answering.** In general, any sort of information retrieval might benefit from a look at the semantic tags; queries of "when" and "how long" are likely to be answered with a constituent marked with a temporal tag, queries of "why" with a purpose constituent, "where" with a location, and so on.

**Machine translation.** Function tags would assist in a number of places in the MT process, we think. When translating prepositions, many times there is a different translation according to the sense of the preposition—which is often captured by a function tag. For languages with many cases, the case often encodes similar information to the function tag, and thus having function tag information on either the source or the destination language might be helpful.

$$
\begin{aligned}
\text{(with-null) Accuracy} &= \frac{\text{correctly tagged} + \text{correctly non-tagged}}{\text{all nonterminal constituents}} \\
\text{Precision} &= \frac{\text{correctly tagged}}{\text{tagged by tagger}} \\
\text{Recall} &= \frac{\text{correctly tagged}}{\text{tagged in gold standard}} \\
\text{F-measure} &= \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}
\end{aligned}
$$

Figure 2.3: Measures of accuracy

## 2.4 Evaluating performance on the task

We can imagine a number of different metrics for determining the correctness of a function tagging. The definition we chose is to say a constituent had been correctly function tagged if there exists in the correct parse a constituent with the same start and end points, label, and function tag (or lack thereof). Since we treated each of the four function tag categories as a separate feature for the purpose of tagging, evaluation was also done on a per-category basis.

The denominator of any accuracy measure should be the maximum possible number we could get correct. In this case, that means excluding those constituents that were already wrong in the parser output; the parser we used (Charniak, 2000), for example attains 89% labelled precision-recall, which means that roughly 11% of the constituents are excluded from the function tag accuracy evaluation. (For reference, we will in several cases also include the performance of our function tagger directly on treebank parses; it turns out not to matter much, and we will discuss the differences as we come to them.)

Another consideration is whether to count non-tagged constituents in our evaluation. Internally, we treat the lack of a tag as the presence of a null tag, so it might make sense to have an overall accuracy measure that includes all constituents in the denominator and all correctly-tagged and correctly-non-tagged constituents in the numerator. However, because the majority of Penn treebank constituents have no tag, in most reasonable situations this would unreasonably inflate the numbers. We will report this 'with-null accuracy' number in a few cases, but for the most part we will just omit it. (Note, though, that for function tag systems like that in (Brants et al., 1997), where there is no null tag, a single accuracy evaluation such as this one is all that is required.)

A much better measure of performance is to just consider those constituents that have some function tag on them. Of course, the number of tagged constituents in the tagger output is likely to be different from that in the gold standard, so we report two numbers. The *precision* is, of the number the tagger guesses, how many it gets correct. This measure is the opposite of the false positive rate. The *recall* is, of the number in the gold standard, i.e. the number it *could* have gotten, how many did the tagger find. This, then, is the opposite of the false negative rate. Both these are shown in Figure 2.3.

The last item in that figure is the *F-measure*, which is a sort of average of the other two. Specifically, it is the harmonic mean, which can also be written as

$$\frac{2}{F} = \frac{1}{P} + \frac{1}{R}.$$

Its use within the AI and machine learning communities is designed to penalise widely divergent precision/recall values. If two values are close, then their harmonic mean will be close to halfway between them; if they are far apart, it will be much closer to the lower of the two. We might imagine a system that spent a huge amount of effort to discover just one tag with maximum confidence, and then left everything else untagged; such a system would have 100% precision but nearly 0% recall, and hence its F-measure would be, not 50%, but nearly 0%. Some versions of the F-measure give an $\alpha$ scaling factor to lend more weight to either the precision or the recall, but we will be using this evenly-weighted formula.

But what exactly is the input to our algorithm? We will be assuming that we have the syntactic structure of a sentence available to us, so we will need parse trees. In the testing, then, it would be easy to simply take the Penn treebank trees, function tag them, and compare them to the hand-annotated versions. However, this is not the most realistic metric, at least not by itself. When a function tagger is actually used, its input will need to come from a parser, and the input will therefore not be perfect. It makes sense, then, to use the output of a high-quality machine parser, rather than the gold-standard parses.

On the other hand, if we aren't careful, this could impose an artificial ceiling on our results. That is, if we take for our denominator the *total* number of function tagged constituents, and we use a parser that has 90% accuracy, the best we can hope to do with our function tagger is just 90%. Generally, then, we will take for our denominator the number of function-tagged constituents that have been parsed correctly in the input. This does give our algorithm a theoretical maximum performance of 100%. (Why not just go back to the gold standard parse, then, if we throw out all the bad constituents anyway? It turns out to make a difference, which will be discussed in Chapter 7.)

While we are testing large numbers of systems on the development corpus, we will sometimes use the gold standard input and sometimes the machine-parsed, as convenient, but the final results will be reported for machine-parsed input, with the bad constituents thrown out. In order to better compare against certain other systems, in Chapter 7, we will also give estimated numbers for our performance including those bad constituents, though as we will explain there, this is really more of a lower bound.

Having established our evaluation schema, we now turn to one of the basic concepts underlying all the systems we will be testing.

# Chapter 3

# Features

A *feature* is nothing more than a question that can be asked. The sorts of features that we are interested in are features of sentence constituents, and tend to be questions like “what is the label of this constituent?” or “what word is the head of the parent of this constituent?” or something along those lines.

Use of the word ‘feature’ in approximately this sense was in use by the mid-90s; it is used thus in (Magerman, 1995) and (Berger et al., 1996), but is in any case now standard within the NLP community for maximum entropy models and some other applications. Older papers on maximum entropy (Jaynes, 1957; Guiaşu and Shenitzer, 1985) just use the term ‘function’, and papers on perceptron models tend not to talk about the individual features at all, referring to them in aggregate. We will nevertheless use the term throughout, for consistency.

## 3.1 Binary vs. multivalued features

Strictly speaking, the algorithms we will be using require that a feature be binary; that is, a yes/no question. Internally, the answer to the question—the value of the feature—will be interpreted numerically, and thus using multivalued features would give rise to strange notions like S being “twice as much as” VBN, whatever that might mean.

However, we can emulate multi-valued features using binary features. Each multi-valued feature with $n$ possible values is emulated with $n$ binary features, of which exactly one has the value 1 and the rest 0. For example, a feature like “what is the label of this constituent?” is changed to the sequence of features “is the label of this constituent NP?”, “is the label of this constituent S?”, and so on.

Conceptually, however, it is very useful to talk in terms of the multi-valued features, and hence we will abuse the terminology slightly and use ‘feature’ to refer sometimes to multi-valued and sometimes to binary features; the context should generally disambiguate.

## 3.2 Some possibly useful features

In this section we describe the features we will be using in our systems in Chapters 4 and 6.

### 3.2.1 Label

The first and most obvious feature to make use of is the label of the constituent. Particularly for the syntactic tags, this is probably the single most important feature; a VP is essentially never going to get any of the syntactic tags, an ADJP is unlikely to be anything other than a PRD (or nothing), and the only function tag that really gets applied to PP constituents is PUT (and that only rarely).

### 3.2.2 cc-Label

This feature normally is identical to the regular Label feature; but if a phrase is comprised of the conjunction of two or more noun phrases, rather than giving the true label NP for the whole phrase, this feature will give a special label CCNP to the phrase. Similarly for verb phrases (CCVP), and for clauses (CCS).

Knowing the value of this feature for the parent of a given constituent was shown in (Charniak, 2000) to be useful for parsing, so we decided to try it on function tags as well.

### 3.2.3 Head

Every linguistic phrase has one word, the *head*, that can be described as the "most important" word in that phrase, syntactically speaking. In a dependency grammar, the head is the word from which all the other words in a phrase ultimately depend. In a categorial grammar, the head-containing portion of a phrase is the functor, and to find the head we can recursively descend through the functors until we reach a single word. The idea is that all but one of the words in any given phrase is either an adjunct or a complement to some other word within that phrase; the one exception is the head of the phrase, which is adjunct or complement to some other word *outside* the phrase.

The head has definitely proven useful in parsing; David Magerman's work (1995) using lexical information showed improvement over the existing state of the art, and Michael Collins (1996) showed that substantial improvement—on the order of 9%—could be attributed to head information alone. So, too, with function tags. The constituent label is not nearly fine-grained enough to distinguish many of the different function tag cases; for instance, nearly all of the semantic tags regularly occur on prepositional phrases. Without some amount of lexical information, the problem is hopeless.

Within the linguistic formalism of the Penn treebank (such as it is), the marking of the head is not inherent to the parse structure. As a result, we need an algorithm to determine the head of any given parsed phrase. The algorithm used by the Charniak parser, and by the function taggers presented here, is shown in broad pseudocode in Figure 3.1 on the next page. The functions goodHead and okayHead return sets of constituents that can be head-containing children, and are defined in Figure 3.2. The algorithm does not actually return the head of the constituent $t$, but rather the

```
findHeadContainingChild (t)
    if ∃c ∈ children(t) s.t. label(c) ∈ goodHead(label(t))
        if label(t) = PP
            return leftmost such c
        else
            return rightmost such c

    else if ∃c ∈ children(t) s.t. label(c) = label(t)
        return leftmost such c

    else if ∃c ∈ children(t) s.t. label(c) ∈ okayHead(label(t))
        return rightmost such c

    (* we're in a strange constituent; back off as best we can *)
    else if ∃c ∈ children(t) s.t. terminal(c) & ¬ punctuation(c)
        return rightmost such c
    else if ∃c ∈ children(t) s.t. ¬ terminal(c) & label(c) ≠ PP
        return rightmost such c
    else if ∃c ∈ children(t) s.t. label(c) = PP
        return rightmost such c
    else
        return rightmost c ∈ children(t)
```

Figure 3.1: Head-finding algorithm

goodHead:

ADJP ⇒ JJ JJR JJS
ADVP ⇒ RB RBB
LST ⇒ LS
NAC ⇒ NNS NN PRP NNPS NNP
NX ⇒ NNS NN PRP NNPS NNP
NP ⇒ NNS NN PRP NNPS NNP $ POS
PP ⇒ IN TO RP
PRT ⇒ RP
S ⇒ VP
S1 ⇒ S
SBAR ⇒ IN WHNP
SBARQ ⇒ SQ VP
SINV ⇒ VP
SQ ⇒ MD
VP ⇒ VB VBZ VBP VBG VBN VBD TO MD
WHADJP ⇒ WRB
WHADVP ⇒ WRB
WHNP ⇒ WP WDT WP$
WHPP ⇒ IN TO

okayHead:

ADJP ⇒ VBN RB
NAC ⇒ NP CD FW ADJP JJ
NX ⇒ NP CD FW ADJP JJ
NP ⇒ CD ADJP JJ NX
QP ⇒ $ NN
S ⇒ SINV SBARQ X
PRT ⇒ RB IN
SBAR ⇒ WHADJP WHADVP WHPP
SBARQ ⇒ S SINV X
SINV ⇒ SBAR
SQ ⇒ VP

Figure 3.2: Which constituent types make good heads vs. tolerable heads

head-containing child; the head of $t$ is then recursively defined as the head of $t$'s head-containing child.

### 3.2.4 Head part-of-speech (POS)

Sometimes the head proves to be a little too fine-grained. Eugene Charniak (2000) showed that for the parsing problem, a small but significant improvement—about 2%—could be achieved by using the part-of-speech of the head as a backoff feature for when the head itself was unknown or rare.

### 3.2.5 Alt head

The semantic function tags are largely applied to prepositional phrases; and those phrases are not easily distinguished solely by their head. The preposition 'in' could head a temporal phrase, or a locative one; 'to' could indicate a direction, or a purpose. In order to properly tell these cases apart, we really need access to the object of the preposition. The 'alt head' grants us this access: it is the head of the object of a prepositional phrase (and undefined for other sorts of constituents). We hoped for a great deal of improvement on the semantic tags due to this feature.

In fact, this could be thought of as a specific example of the lexical-functional head distinction(Grimshaw, 1990)—"alt heads" being specifically the lexical heads of prepositional phrases. We did not implement the more general "lexical head" feature.

### 3.2.6 Alt head POS

While the alt head's part of speech won't help much in distinguishing 'in Chicago' from 'in March', it might give us some assistance with 'to Chicago' and 'to win'.

### 3.2.7 Function tags

The function tags are themselves features, of course. Obviously we cannot use a function tag as a feature for guessing itself, but it's quite possible that using one category of function tag may help us in the prediction of another. For instance, subjects are rather unlikely to be modifiers of time, place, or anything else; and a PP marked with a semantic tag is unlikely to contain a logical subject (LGS).

### 3.2.8 Label clusters

Data for labels like NP can often be found without much difficulty, but some types of constituent are considerably less frequent, such as WHNP. In some contexts, these less frequent constituents will act just like their more common relatives. Furthermore, at the POS tag level, there are often multiple tags to represent some syntactic or morphological distinction that may not be relevant for our purposes. And finally, the linguistic formalism in use for the Penn treebank has disjoint label sets

| | | |
|---|---|---|
| ADJP JJ JJR JJS WHADJP | IN TO | CONJP CC |
| NP NN NNS NNP NNPS NX WHNP | DT WDT | INTJ UH |
| ADVP RB RBR RBS RP WRB PRT WHADVP | PRP WP | LST LS |
| VB VBD VBG VBN VBP VBZ | PRP$ WP$ | PP WHPP |
| S SBAR SBARQ SINV SQ | | |

Figure 3.3: Label clusters

for nonterminals (phrases) and terminals (words); yet there is often a strong correspondence between a phrasal label and one or more POS tags (as with ADJP (adjectival phrase) and JJ (adjective)).

To address these observations, we have manually created a number of label clusters, to collapse distinctions that may not be relevant in all cases. These clusters are shown in Figure 3.3. (Those labels not listed are each in their own, singleton, cluster.) We hope that inclusion of the label cluster feature will increase performance on the function tagging task in cases of sparse data. A secondary hope is that *replacing* the label feature with the label cluster will maintain equivalent performance, at least in certain cases, making the algorithm more general.

### 3.2.9 Word clusters

The worst of the sparse data problems are engendered by individual lexical items themselves; with nearly 40,000 words it is simply not possible to have solid data on all of them in all contexts. Rather than just ignoring the rare words entirely, though, it seems like it would be useful to group them to fend off the sparse data problems. We ran an algorithm to group all words with a given part of speech into a relatively small number of clusters, each of which had a unique identifier; this identifier is then the value of this word cluster feature.

## 3.3 Making use of relatives

Each of the features listed in the previous section can apply not only to the constituent being tagged, but also to its various relatives, since we accept parsed sentences as input to our systems. The most useful relative will obviously be the parent, but we also included features of the grandparent and both adjacent siblings, as well as some features of more distant siblings. Different such features will be useful in different situations: the parent will come in handy for tags like LGS, where the parent needs to be a prepositional phrase headed by 'by', while the right sibling may play an important role in choosing whether a constituent is the SBJ (subject), since there can often be multiple noun phrases sibling to the verb phrase in a sentence, in appositive or modificational constructions, and they can be partially distinguished by their linear positioning in the sentence (relative to each other, to commas, and to the verb phrase).

# Chapter 4

# Feature trees

In this chapter we will describe the basic algorithmic mechanisms introduced in Charniak's paper *A maximum-entropy-inspired parser* (2000), and their application to and performance on the function tagging problem. As the term "maximum-entropy-inspired" is rather unwieldy as a term of art (and not particularly evocative of the actual algorithm), we have adopted the descriptor "feature trees", referring to the core structures involved.

## 4.1 A brief history and overview of maximum entropy

The driving principle behind maximum entropy models is that when assigning probability distributions based on limited information, if we have no information on the relative likelihood of two events then we should assign them equal probability. More generally, given the information that we *do* have, we want to assign the most uniform probability distribution that is consistent with that information. As pointed out in (Berger et al., 1996) and elsewhere, this is not a new idea, arguably dating back to William of Ockham or earlier. The Principle of Indifference (or Insufficient Reason) has been given in various forms, of which the earliest clear and general statement was probably by Johannes von Kries (1886, p. 6):

> Two or more cases are to be regarded as equally possible, when in their respective circumstances we can find no reason to maintain one as possibly more probable than some other.

although it has a documented pedigree stretching back at least to Jacques Bernoulli.[1]

When we do have information to distinguish the probability of two events, we won't want to consider them equally likely, of course. More importantly, if we have *many* possible events, and *some* information about their respective likelihoods, we will need some way to calculate a probability distribution that will make them as equally likely as possible, within the constraints of that

[1]The pedigree includes Laplace, who often gets credit for the whole thing. For an interesting digression on the history of this principle, see Appendix D of this thesis.

information. In order to measure this notion of "as equally likely as possible", we will turn to the *entropy* measure $H$, introduced by Shannon (1948, Section 6):

$$H = -K \sum_{i=1}^{n} p_i \log p_i \,. \tag{4.1}$$

(for some constant $K$, denoting a scaling factor; we will assume $K = 1$). It measures how uncertain we are about a given probability distribution $p$. For example, if the log is taken base 2, and $p$ is equally distributed over two choices, the entropy $H$ will be exactly 1. If, however, $p_1$ is 0 and $p_2$ is 1—meaning we are certain that the second choice will occur—then the entropy of this distribution will be exactly 0—indicating that there is no uncertainty about the outcome.

Now that we have a concrete measure of the uncertainty present in any given probability distribution, we might recast the Principle of Indifference as indicating that we should choose a probability distribution that has no certainty that is unwarranted; in other words, the distribution that has the greatest *un*certainty, or entropy, given our constraints. Edwin Jaynes (1957) made just that observation, and named it the principle of *maximum entropy.* For a more complete discussion of the underlying mathematics, one can refer to the original Jaynes paper or the slightly more accessible (Guiaşu and Shenitzer, 1985).

In Equation 4.1 we have an expression for the entropy of a distribution $p$. Following (Berger et al., 1996), let us consider a conditional distribution $p(y|x)$, the probability of some event $y$ given conditioning information $x$; then we can talk about the entropy of the distribution with respect to a single example $x$:

$$H_{p,x} = -\sum_{y} p(y|x) \log p(y|x) \tag{4.2}$$

Then it is fairly straightforward to extend this to a definition of the entropy of $p$ with respect to a whole corpus of examples $x$; we calculate the entropy relative to each individual example, then sum them, weighting according to the (observed) frequency of each example $x$.

$$\begin{aligned} H_p &= -\sum_{x} \hat{p}(x) \sum_{y} p(y|x) \log p(y|x) \\ &= -\sum_{x,y} \hat{p}(x) p(y|x) \log p(y|x) \end{aligned} \tag{4.3}$$

Now, in principle, we can propose any $p$ at all, evaluate its entropy with respect to a training corpus, and compare it to other proposed $p$ distributions, finally selecting the one with the greatest entropy. In reality, of course, there exists an infinitude of possible distributions, and we can't hope to evaluate them all. However, we can restrict ourselves to $p$ of the form

$$p_\lambda(y|x) = \frac{1}{Z_\lambda(x)} e^{\sum_i \lambda_i f_i(x,y)} \tag{4.4}$$

and it turns out that we can reduce the problem of finding the most entropic $p$ to a somewhat more feasible optimisation problem on the $\lambda_i$ values. (A justification for this restriction and explanation

of this reduction is available in (Berger et al., 1996) and earlier papers.) Furthermore, once a model is trained, and the $\lambda$ values are known, calculation of this probability is very easy. If the goal of the calculation is merely to score different $y$ values, the $Z$ factor—which serves to normalise the value into a probability distribution summing to 1—can be ignored, along with the exponentiation, without changing the relative ranking of the $y$ values. The $f(x, y)$ is a feature value (generally binary), and the lambdas are precalculated, making the remaining scoring task equivalent to calculating a few simple dot products.

In addition to machine translation, which was maximum entropy's entrée into the field of NLP (Berger et al., 1996), maximum entropy models have been proposed for other NLP tasks, including parsing (Ratnaparkhi, 1999). The Ratnarparkhi model uses the model to tie together and impose statistics onto a three-pass parser that first tags, then chunks, then combines the existing chunks using a shift-reduce parser.

## 4.2 From interpolation to feature trees

A simple, and now standard, technique for smoothing a probability model is known as *held out interpolation*, introduced in (Jelinek and Mercer, 1980). When one has a probability distribution with a great deal of conditioning information, it is likely that that distribution will sometimes become unreliable due to sparse data—if a given set of conditioning events has only been seen a few times, the empirical distribution it engenders is likely to vary considerably from the platonic 'true' distribution. To combat this problem, we "fall back" to a distribution with fewer conditioning events: less specific, but more likely to be close to the 'true' distribution. There is no reason to limit the fallback to a single level, either; we can give consideration even to the most general distribution available. The extent of the reliance on less specific distributions is controlled by an interpolation factor $\lambda$:

$$p(y|x_1, \ldots, x_n) = \sum_{i=1}^{n} \lambda_i(x_1, \ldots, x_n)\hat{p}(y|x_1, \ldots, x_i) \quad (4.5)$$

In this, its most general form, each $\lambda_i$ is a function of all the conditioning information, permitting the final distribution to rely more on specific distributions for those conditioning environments that are common, and more on the general distributions for the rarer conditioning environments.

The key observation made by Charniak (2000) was that where an interpolation model would use interpolation factors on several probabilities to calculate a final distribution, a maxent model with appropriate features might make the exact same calculation. Consider a case where an event $y$ is conditioned on three other events $x_1, x_2, x_3$; and in this particular case their values are $a$, $b$, $c$. In an interpolation model we thus have

$$p(y|a, b, c) = \lambda_1(a, b, c)\hat{p}(y|a, b, c) + \lambda_2(a, b, c)\hat{p}(y|a, b) + \lambda_3(a, b, c)\hat{p}(y|a)\,. \quad (4.6)$$

But let's say we've trained a maxent model with features for every value of $x_1$, every pair of values $x_1, x_2$, and every triple of values $x_1, x_2, x_3$. In this case, most of the features will have values of zero,

leaving the exponent in the probability formula (cf. (4.4)) to be

$$\lambda_{y,a,b,c} f(y,a,b,c) + \lambda_{y,a,b} f(y,a,b) + \lambda_{y,a} f(y,a)\,. \tag{4.7}$$

Strikingly similar in both form and effect; this invited further comparison.

Charniak first observed that since the maxent case was additive in the exponent, it could be rewritten as a multiplicative sequence

$$p(y|a,b,c) \quad = \quad g_0(y,a,b,c) g_1(y,a,b,c) \cdots g_j(y,a,b,c)\,, \tag{4.8}$$

where each $g_i$ could be rewritten as $e$ raised to a power containing a lambda and a feature function.[2] Each term of this product is zero if the relevant feature is independant of the event being predicted, less than one if the feature contraindicates the predicted event, and greater than one if the feature correlates positively with the predicted event; and thus we can think in a convenient way about what effect each feature has on the final probability. Would it be possible to take the basic model, interpolated or not, and give it this sort of incremental property?

It would, and with interesting effect. As a first pass, we ignore the smoothing weights and just rewrite the base distribution $p(y|x_1,x_2,x_3)$ as follows:

$$p(y|x_1,x_2,x_3) \quad = \quad \hat{p}(y) \frac{\hat{p}(y|x_1)}{\hat{p}(y)} \frac{\hat{p}(y|x_1,x_2)}{\hat{p}(y|x_1)} \frac{\hat{p}(y|x_1,x_2,x_3)}{\hat{p}(y|x_1,x_2)} \tag{4.9}$$

Due to cancellation of terms, this equation is trivially true. But it has the property noted in the previous paragraph: after the first term (which is a probability), all subsequent terms either increase or decrease the probability based on the evidence provided by the newest feature.

Arranging the terms in this fashion highlights another fact as well. In each case, as we add a new feature to consideration, we are continuing to condition on all the previous conditioning information. That is surely the correct thing to do sometimes, but perhaps not always. For example (argues Charniak), perhaps we want "to condition on the parent's lexical head without conditioning on the left sibling, or the grandparent label." If, in our example, we feel that $x_2$ and $x_3$ are independent with respect to their predictive power on $y$, then we would drop $x_2$ from the final term of (4.9), yielding the following:

$$p(y|x_1,x_2,x_3) \quad = \quad \hat{p}(y) \frac{\hat{p}(y|x_1)}{\hat{p}(y)} \frac{\hat{p}(y|x_1,x_2)}{\hat{p}(y|x_1)} \frac{\hat{p}(y|x_1,x_3)}{\hat{p}(y|x_1)} \tag{4.10}$$

The first thing to notice is simply that the equation no longer reduces to a tautology, even in this non-smoothed form. More specifically, though, we not only removed $x_2$ from the numerator of the final term, but from the denominator as well. This is consonant with our notion of each term being a separate greater- or less-than-one multiplier; and representing the marginal "additional information"

[2] Now is probably a good time to warn against confounding the two types of lambda in this discussion. In an interpolation model, $\lambda$ represents a factor that is multiplied against an empirically observed probability, acting as a weight on that probability. In a maxent model, however, the feature function is binary, so the $\lambda$ carries the entire responsibility for "how does this feature affect the final probability"; but neither the feature, nor the $\lambda$, nor their product can be meaningfully said to be a probability by itself.

granted by a new feature over some previous estimate. The difference being, it is no longer the immediately previous estimate. Formally, the system is now not a probability distribution, since in most cases the sum of all scores will not be 1; this can be corrected with a normalisation term if a true probability model is desired, although we will omit such a term from the ensuing discussion.

We can think about this change graphically. Under the old system, of (4.9), the calculation can be represented with a chain of nodes, each representing a different term (and feature), and having a sort of predictive dependence on the immediately preceding node; this is shown in Figure 4.1. By contrast, (4.10) can be represented as a tree, as in Figure 4.2, where each node is no longer necessarily linked to the immediately "preceding" node if we have judged it to be predictively independent. (We use this slightly awkward notion "predictive independence" to highlight the fact that these variables may not be—and probably are not—independent in the probabilistic sense.)

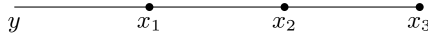


Figure 4.1: A feature chain

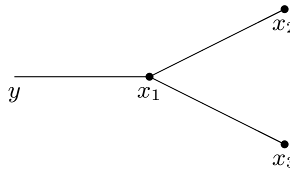


Figure 4.2: A feature tree

Now we can return to the question of how to smooth these new feature-tree-based probabilities. In a totally unsmoothed world, many of the terms in (4.10) would still cancel out, reducing to

$$p(y|x_1, x_2, x_3) \quad = \quad \frac{\hat{p}(y|x_1, x_2)\hat{p}(y|x_1, x_3)}{\hat{p}(y|x_1)}\,. \tag{4.11}$$

Of course, we will want each of the component probabilities to have some influence on the final result (without losing our greater- or less-than-one property). The solution: separately smooth each of the component probabilities, making this equation

$$p(y|x_1, x_2, x_3) \quad = \quad \frac{p(y|x_1, x_2)p(y|x_1, x_3)}{p(y|x_1)}\,; \tag{4.12}$$

yes, that is the same formula *sans* the hats over the $p$ on the right side: these are just the smoothed probabilities for this distribution.

In principle, any smoothing will work with this schema; the one we have used is a form of interpolation that is calculated recursively:

$$p(y|b_1, \ldots, b_n) \quad = \quad \lambda_n(b_1, \ldots, b_n)\hat{p}(y|b_1, \ldots, b_n) + (1 - \lambda_n(b_1, \ldots, b_n))p(y|b_1, \ldots, b_{n-1}) \tag{4.13}$$

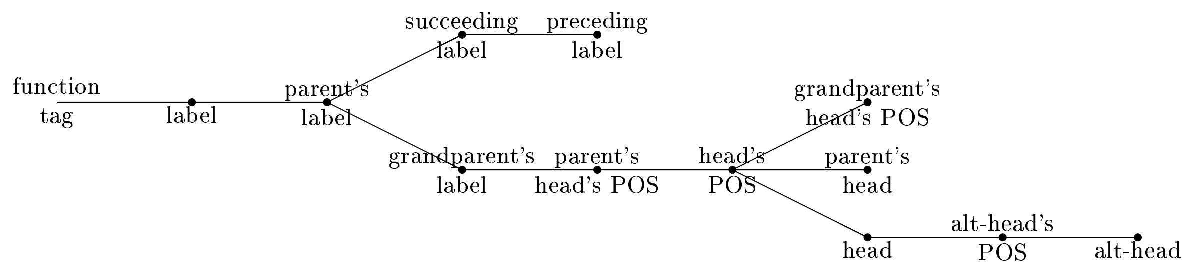


Figure 4.3: The generic feature tree used to guess function tags in the original paper

Note the location of the hats, and the extent of the $b$ sequences: what this formula does is take a relatively specific empirical distribution, and mixes it in with the pre-existing, less specific, smoothed distribution.

The chief advantage of the feature tree system, as compared to the older feature chain system (i.e. plain interpolation), is that it is able to make use of more information in the face of sparse data. Especially when lexical items are involved, the data frequently become too sparse before all the desired features have been considered; a lexical feature will usually be the last one in a chain to have influence, simply because there are on the order of 40,000 words. In a feature chain, this means that we are effectively limited to a single lexical feature, which will be the last one considered. In a feature tree, however, each lexical item is permitted to have its own branch on the tree, thus mitigating if not entirely negating this effect.

## 4.3 Experimental design

As is standard with parsing, we have divided the Penn treebank II Wall Street Journal corpus into training, development, and testing sections. For each feature tree that we wish to test, we first run a pass over the training sections (2–21) to gather the statistics for the $\hat{p}$ distributions. The smoothing parameters $\lambda$ are then estimated based on the held-out development section (24); and then performance is evaluated. For the purposes of seeking out the best feature trees, we re-used section 24 for testing, then selected just the best feature trees and reported final results on the true testing section (23).

## 4.4 Results

In our first reported results on this work (Blaheta and Charniak, 2000), we gave results derived from a single feature tree, shown in Figure 4.3. The version of those results we give in Table 4.1 on the facing page is modified slightly from the paper; first, those results were reported only for sentences shorter than forty words (these are for all sentences), and second, those results included performance on the CLR tag, which we now ignore (cf Section 2.1.1).

| | With-null | ———No-null——— | | |
|---|---|---|---|---|
| | Accuracy | Precision | Recall | F-measure |
| Syntactic | 98.820% | 94.803% | 95.509% | 95.155% |
| Semantic | 97.069% | 80.458% | 79.169% | 79.148% |
| Topicalisation | 99.924% | 93.724% | 93.724% | 93.724% |
| Miscellaneous | 99.782% | 75.676% | 26.667% | 39.437% |
| | | | | |
| Overall | 98.899% | 87.927% | 88.631% | 88.626% |

Table 4.1: Performance of tree in Figure 4.3

After this initial set of results, we tried to find the best trees for the job, permitting different trees for each function tag category. Starting with a feature tree of zero nodes (i.e. just the prior), we repeatedly perturbed it by one of four operations:

- adding a new feature at the end of one branch,
- swapping two features in the middle of the tree (e.g. changing a tree $a{-}b{-}c{-}d$ into $a{-}c{-}b{-}d$),
- moving one of the features at the end of one branch onto a different (possibly new) branch of the tree, or
- (occasionally) inserting a new feature into the middle (usually this was only done when we looked at a tree and saw a "head" feature without its corresponding "head's POS" feature).

At all points we kept the several feature trees that performed best for each category on the development set (section 24 of the Penn treebank).

Since the purpose was primarily exploratory, the process was never fully automated, so the choice of "the several best" was to some extent subjective, and sometimes one better tree would be thrown out in favour of another, if the better tree was very similar both in form and in performance to an even better tree. In this way we hoped to avoid getting stuck in local maxima. Another factor in avoiding local maxima was that all categories were tested for each feature tree; this had the effect of retaining a number of fairly (increasingly) diverse trees in the pool for each category. Continuing research in this direction could certainly involve automating this process using any of a number of standard search algorithms.

A selection from the thousands of runs is given in Table 4.2 on the following page. In the left column is a sequence that identifies the feature tree used to calculate the statistics for those runs, which can be read as follows: each number represents a feature, as enumerated in Table 4.3 on page 27. Sequences of only numbers represent feature chains; occurrences of the letter 'B' signify backing up one space on the current path and attaching the next feature there. (Two Bs indicate attaching the next feature two nodes back, and so on.) For reference, the final line of the table represents the generic feature tree of Figure 4.3 on the facing page. Superscripts mark trees and values that will be discussed later in this chapter or in Chapter 7.

| | Syntactic | Semantic | Topicalisation |
|---|---|---|---|
| 0 | 00.00 | 00.53 | 00.00 |
| 3 | 40.98 | 52.92 | 00.00 |
| 8 | 79.08 | 00.53 | 00.00 |
| 0/1 | 89.86 | 14.60 | 54.82 |
| 0/8 | 86.24 | 02.49 | 00.00 |
| 0/12 | 89.86 | 14.60 | 82.93 |
| 8/0 | 86.24 | 02.98 | 00.00 |
| 8/1 | 83.73 | 01.44 | 54.41 |
| 8/4 | 80.56 | 05.60 | 72.28 |
| 3/30 | 41.68 | 66.54 | 00.00 |
| 0/1/4 | 93.48 | 31.65 | 87.91 |
| 0/1/7 | 92.49 | 28.70 | 72.39 |
| 0/8/4 | 91.97 | 31.52 | 89.84 |
| 3/30/6 | 73.71 | 74.96 | 25.21 |
| 3/30/7 | 59.85 | 74.51 | 25.21 |
| 3/30/12 | 82.06 | 74.82 | 51.52 |
| 0/1/4/3 | 94.73 | 53.94 | 87.78 |
| 0/1/4/7 | 94.56 | 40.47 | 89.73 |
| 0/1/4/10 | 94.19 | 33.97 | 88.53 |
| 0/8/4/3 | 93.26 | 51.39 | 91.80 |
| 0/8/4/12 | 92.72 | 39.47 | 91.80 |
| 3/30/12/33 | 83.90 | 76.66 | 89.62 |
| 3/30/12/0 | 83.90 | 76.64 | 88.89 |
| 3/30/7/42 | 79.24 | 76.34 | 74.71 |
| 3/30/7/33 | 61.10 | 76.33 | 54.43 |
| 0/1/4/10/7 | 95.21 | 39.38 | 89.25 |
| 0/1/4/10/3 | 95.19 | 49.69 | 89.13 |
| 0/1/4/3/7 | 95.13 | 54.19 | 87.78 |
| 0/1/4/7/10 | 95.15 | 41.62 | 89.73 |
| 0/1/4/7/9 | 95.03 | 41.55 | 89.73 |
| 0/1/4/7/3 | 95.18 | 52.14 | 90.71 |
| 0/1/4/10/7 | 95.21 | 39.38 | 89.25 |
| 3/30/12/33/4 | 86.23 | 77.93 | 89.13 |
| 3/30/7/33/4 | 74.38 | 77.63 | 76.67 |
| 3/30/12/0/11 | 84.96 | 77.19 | 89.40 |
| 3/30/12/0/7 | 85.58 | 77.19 | 90.11 |
| 3/30/12/33/6 | 85.66 | 77.17 | 87.43 |
| 0/1/4/10/3/7 | 95.58 | 49.72 | 89.62 |
| 3/30/12/0/7/4 | 86.52 | 78.18 | 89.62 |
| 3/30/12/33/4/38 | 86.23 | 77.63 | 89.13 |
| 3/30/12/33/4/2 | 86.23 | 77.63 | 89.13 |
| 3/30/12/33/4/6 | 86.23 | 77.63 | 89.13 |
| 0/1/8/B/4/10/3/7 | 95.62 | 50.59 | 91.49 |
| 0/1/8/B/6/4/10/3/7 | 95.70 | 50.74 | 87.76 |
| 0/1/8/B/6/4/10/7/B/3 | 95.78 | 52.06 | 87.76 |
| 0/1/8/B/6/4/10/7/B/2/3 $^{SY}$ | 95.80 | 52.12 | 88.66 |
| 3/29/30/12/0/7/4 $^{SM}$ | 86.49 | 78.80 | 89.62 |
| 2/3/29/30/12/0/7/4 | 85.25 | 78.62 | 90.91 |
| 2/3/29/30/B/B/B/12/0/7/4 | 94.86 | 77.82 | 93.95 |
| 0/2/3/29/30/B/B/B/12/7/4 | 94.94 | 79.00 | 92.59 |
| 0/3/29/30/B/B/B/12/7/4 | 94.80 | 78.06 | 93.09 |
| 0/1/8/B/4/10/2/3/B/7 | 95.66 | 52.21 | 92.47 |
| 0/1/8/7/B/B/9/6/2/10/B/4/B/3/29/30 | 94.58 | 73.58 | 92.59 |

Table 4.2: F-measure from various test runs on Section 24

| | | | |
|---|---|---|---|
| 0 | label | 9 | grandparent's label |
| 1 | parent's label | 10 | grandparent's head's POS |
| 2 | head's POS | 11 | grandparent's head |
| 3 | head | 12 | parent's cc-label |
| 4 | parent's head | 29 | alt-head's POS |
| 6 | parent's head's POS | 30 | alt-head |
| 7 | left sibling's label | 33 | label backoff |
| 8 | right sibling's label | 38 | parent's head's cluster |

Table 4.3: Key to numeric feature IDs in Figure 4.2 on the preceding page

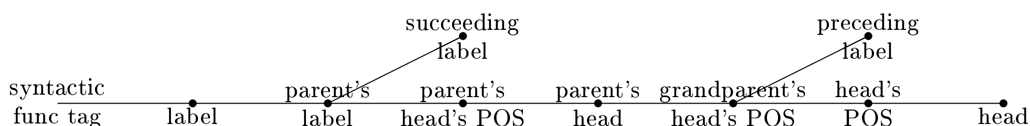


Figure 4.4: The feature tree used to guess syntactic tags

The final trees we arrived at for the syntactic and semantic tags can be seen in Figures 4.4 and 4.5, marked as SY and SM respectively in Table 4.2. This method failed to improve on the generic tree for use in topicalisation; and the miscellaneous category was optimised including the CLR tag, and so the results are no longer relevant, so we only give new results on the final test set for the first two categories. The results of using these new trees on the true test section, Section 23, are given in Table 4.4 on the following page.

## 4.5 Incorporating function tagging into parsing

Because this algorithm for calculating function tags is based on an algorithm originally conceived for parsing sentences, it seemed as if it would be straightforward to integrate the two—maintaining a full list of candidate parses, with their probability scores, and function tagging all of them, returning the combination of parse and function tags that yielded the highest aggregate probability score. This, we hoped, might yield two benefits. The first was that it would be an integrated system, convenient for maintenance, modification, and (not least) use in language modelling. The second was that it would actually improve parsing.

Our reasoning went something like this: sometimes, the parser comes up with the wrong answer, but the right answer would have been its second choice. In at least some of these cases, the function

Figure 4.5: The feature tree used to guess semantic tags

| | With-null | ———No-null——— | | |
|---|---|---|---|---|
| | Accuracy | Precision | Recall | F-measure |
| Syntactic | 99.025% | 96.455% | 95.328% | 95.888% |
| Semantic | 97.594% | 86.693% | 80.283% | 83.365% |

Table 4.4: Performance of new trees on Section 23

| | |
|---|---|
| Parse probabilities: | |
| $p$(badparse) | 60% |
| $p$(goodparse) | 40% |
| Probabilities of correct function tag $f$: | |
| $p(f \mid \text{goodparse})$ | 80% |
| $p(f \mid \text{badparse})$ | 30% |
| Joint probabilities: | |
| $p(f \;\&\; \text{badparse})$ | 18% |
| $p(f \;\&\; \text{goodparse})$ | 32% |

Figure 4.6: An example illustrating why integrating systems might help

tag associated with this constituent becomes much more likely when the correct parse is selected. The integrated parser/function-tagger will then select the correct parse. Some example scores illustrating this are given in Figure 4.6: the good parse by itself gets a lower probability score, but taken together with its function tag it receives a higher score.

The good news is, integrating the two systems is possible, and yields a language model. However, it turns out that integrating the systems doesn't significantly affect parsing one way or another. As is so often the case, the cause seems obvious in retrospect: in looking to the correct function tag to rescue the probability, we haven't considered all the competition. In the final ranking, we consider not only the correct function tag, but *all* function tags; and if the correct function tag scored low enough to matter on the bad parse, some other function tag is likely to have scored much better, as in Figure 4.7 on the facing page.

It is certainly possible to devise examples where, even considering all function tags, integrating the function tagging improves the parsing. Such an example can be seen in Figure 4.8 on the next page. These cases are characterised by three main rules:

1. The correct parse must have a score close to the top score
2. The correct parse must have a clear and obvious function tag, with a probability near 100%
3. The high-scoring parse's function tag mustn't be much more likely than the alternatives

Actual parse probability scores tend to be several orders of magnitude smaller than those given in our illustrations here, and even parses with "close" scores differ by considerably more than a factor of 1.5. As rule 1 weakens, rules 2 and 3 must strengthen to compensate. In practice, these situations tend not to occur very often.

Parse probabilities:

| | |
|---|---|
| $p$(badparse) | 60% |
| $p$(goodparse) | 40% |

Probabilities of correct function tag $f$ and incorrect function tag $x$

| | |
|---|---|
| $p(x \mid \text{badparse})$ | 70% |
| $p(f \mid \text{badparse})$ | 30% |
| | |
| $p(f \mid \text{goodparse})$ | 80% |
| $p(x \mid \text{goodparse})$ | 20% |

Joint probabilities:

| | |
|---|---|
| $p(x \;\&\; \text{badparse})$ | 42% |
| $p(f \;\&\; \text{goodparse})$ | 32% |
| $p(f \;\&\; \text{badparse})$ | 18% |
| $p(x \;\&\; \text{goodparse})$ | 8% |

Figure 4.7: An example illustrating why integrating systems doesn't help

Parse probabilities:

| | |
|---|---|
| $p$(badparse) | 60% |
| $p$(goodparse) | 40% |

Probabilities of correct function tag $f$ and incorrect function tag $x$

| | |
|---|---|
| $p(x \mid \text{badparse})$ | 55% |
| $p(f \mid \text{badparse})$ | 45% |
| | |
| $p(f \mid \text{goodparse})$ | 95% |
| $p(x \mid \text{goodparse})$ | 5% |

Joint probabilities:

| | |
|---|---|
| $p(f \;\&\; \text{goodparse})$ | 38% |
| $p(x \;\&\; \text{badparse})$ | 33% |
| $p(f \;\&\; \text{badparse})$ | 27% |
| $p(x \;\&\; \text{goodparse})$ | 2% |

Figure 4.8: An example illustrating why integrating systems might still help

# Chapter 5

# Decision trees

Another sort of system that is based on features is the *decision tree*. However, instead of always using the same features of every constituent, a decision tree chooses which features to use questions to ask *based on* the answers to the previous questions. The tree structure here arises differently from that of feature trees: before any questions are asked, there is a single root state, and as questions are asked and answered the process descends through the branches of the tree. At the leaf nodes we find the decision that the tree has made, the value it chooses for the feature in question.

Decision trees have been used in other natural language tasks (Magerman, 1994; Bahl et al., 1989), and seemed like their ability to change the conditioning features based on the values of other conditioning features would come in very handy. For instance, on the one hand a constituent is very likely to be a predicate (PRD) if it is a noun phrase under a verb phrase headed by a form of 'to be', irrespective of what the head or siblings of the noun phrase might be; on the other hand, a noun phrase that is the left sister of a verb phrase is likely to be a subject (SBJ), whatever the verb in the sentence might be.

## 5.1 Decision trees in function tagging

As an initial pass at the problem of generating decision trees for the function tagging task, we decided to use an existing package: Ross Quinlan has implemented an algorithm to grow decision trees in his package `c4.5`. We converted the same training, development, and testing corpora used in the previous chapter into a format readable by `c4.5`, and trained on a million constituents of the training data, testing (for exploratory purposes) on the development corpus. These results are given in Figure 5.1 on the following page, along with the results from our best trees on the development corpus. Note that these numbers for the feature trees are different from those reported in Figure 4.2; this is because those numbers were for the function tagging task performed on machine-parsed sentences, while the numbers reported here are on gold-standard parses.[1]

[1] Which in turn gives rise to two questions: why didn't we use the machine-parsed sentences here? Why didn't we use the gold-standard parses there? For the former, this was meant as an exploratory first pass at decision trees,

| | Precision | Recall | F-measure |
|---|---|---|---|
| **Syntactic** | | | |
| Feature trees | 98.21 | 97.36 | 97.78 |
| Decision trees | 98.75 | 97.55 | 98.15 |
| **Semantic** | | | |
| Feature trees | 80.61 | 76.88 | 78.70 |
| Decision trees | 81.08 | 71.07 | 75.75 |

Figure 5.1: Preliminary decision tree results

These initial results were slightly promising; the syntactic tags did increase by a small amount, and performance on the semantic tags was not hurt much. This may in part be because of a smaller training corpus for the decision trees: we truncated the training at one million example constituents due to memory limitations. (This was before we thought to strip out the terminal constituents, as we do in the next chapter; but doing so does not significantly increase the number of nonterminals the training can handle, presumably because the algorithm was already paying very little attention to the terminal constituents.)

## 5.2 Why we abandoned decision trees

Improving the performance of the decision trees was unquestionably a valid possible research direction. However, moving beyond the initial exploratory tests would have required either writing a decision tree learner from scratch, or acquiring, learning, and modifying the `c4.5` code itself, in order to improve the results and resolve the memory issues. Neither direction would have been impossible, but as we will see in the next chapter, the perceptron algorithm had a very easy initial implementation and a number of straightforward improvements for us to experiment with; and its performance on the initial algorithm was approximately on a par with the decision tree performance and had no memory problems.

We may at some point return to see if decision trees can be usefully brought to bear on the problem, but for now it seemed more sensible to address the perceptron first.

not as the last word on their performance. For the latter, we discuss this decision at greater length in Section 2.4.

# Chapter 6

# Perceptrons

The perceptron algorithm was originally devised nearly a half century ago as a method of explaining how, when presented with various stimuli, neurons can learn them, store them, and later recognise similar stimuli (Rosenblatt, 1958). Later, computer scientists latched onto a form of the perceptron algorithm to perform the same task in artificial intelligence systems, due to the extreme simplicity of the basic algorithm. It was soon discovered, however, that it had some weaknesses that made it difficult to use—in particular, it only worked on problems that were *linearly separable* (about which more later).

Various solutions have been proposed, most of which made for significantly "heavier" algorithms, but recently there has been some work at using algorithms that retain the simplicity, if not the speed, of the original algorithm. It is these that we consider in this chapter.

## 6.1 A high level view

The key intuition behind this algorithm involves imagining that we are training neurons—perceptrons—to recognise specific types of things. In the classic binary form of the algorithm, we train a single perceptron to say "yes, it is an $X$" or "no, it isn't an $X$" about any given stimulus, whether $X$ is *utterance of the word 'can'* or *odour of a skunk* or *image of my grandmother*. The training consists of exposing the perceptron to many stimuli—some $X$, some not $X$—letting it guess the $X$-ness of each, and then correcting it if it is wrong.

Essentially, the perceptron sees each stimulus as a point in a many-dimensional space. It tries to draw a line—or rather, a hyperplane—to separate the space into two halves, the 'yes' half and the 'no' half. As corrections are made, it adjusts the location of this hyperplane in the space. If it guesses 'no', but the correct answer was 'yes', then the hyperplane is adjusted to put the point corresponding to that stimulus on the 'yes' side of the hyperplane, or at least not as far on the 'no' side. (Repeated attention to that same stimulus would eventually move the hyperplane far enough to put the stimulus on the 'yes' side.) If the stimuli are *linearly separable*, and at least one hyperplane can be drawn that puts all the 'yes' stimuli on one side and the 'no' stimuli on the other,

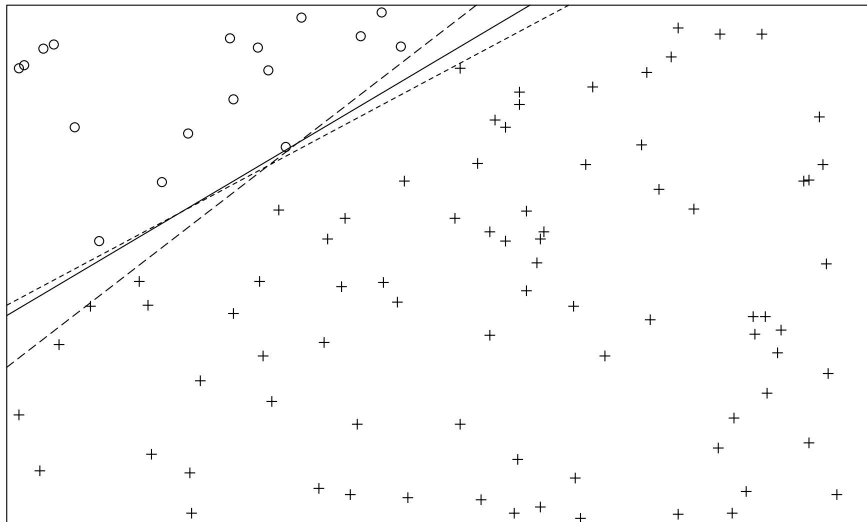

Figure 6.1: A linearly-separable dataset, and some valid separators

the perceptron will find one such hyperplane.

Consider the following two-dimensional example. In Figure 6.1, we have a collection of data in two clearly-defined sets. We can draw a line that completely separates them; three such lines are shown in the figure, and given enough training data, the perceptron algorithm will, eventually, find such a line as well. But we can add just a single data point to that set, indicated by an arrow in Figure 6.2, that makes the set no longer linearly separable: there is no (straight) line that can be drawn across Figure 6.2 that puts all the crosses on one side and all the circles on the other. On such a data set, the perceptron algorithm would never converge. At some point it might draw a line with the only error being the new data point; but then, presented with that data point, it would "correct" the line, and some of the crosses would be on the wrong side of it.

Real-world problems, as it turns out, are rarely completely linearly separable. To humans, this presents little difficulty: presented with a problem like Figure 6.2, most would either just draw a curved line that perfectly separates the two halves of the data, or accept the new circle as an outlier, draw a line that separates the rest of the data perfectly, and be done with it. Both of these tactics have been used in modifying the perceptron algorithm to resolve the convergence problem, and we will discuss them further later in this chapter.

## 6.2 Multi-valued classification

While the classic perceptron algorithm performs a binary classification—'yes' vs. 'no', 'circle' vs. 'cross'—many problems actually involve a decision between more than two choices. In particular, most of the function tag groups have multiple members, so the choice is not "DTV or not DTV?",

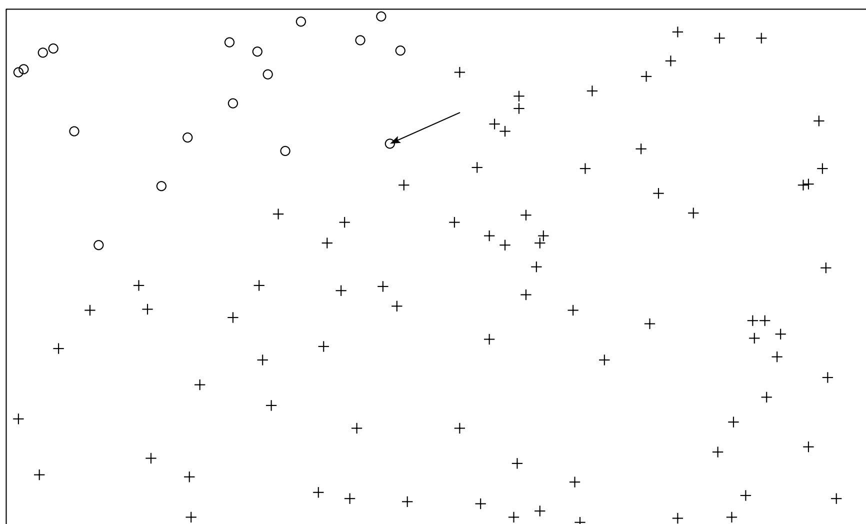

Figure 6.2: A non-linearly-separable dataset

but rather “DTV, or LGS, or ..., or VOC, or no syntactic tag at all?” Hence the problem becomes a little more complicated than training a single perceptron to say ‘yes’ or ‘no’.

Not much more complicated, however. For the multi-valued classification problem, we can imagine a group of $m$ agents—one for each possible output value—each trained to be an expert at guessing whether to assign its output value to a given stimulus. That is, for any given stimulus, a given agent can assign its output value with high confidence, or with low confidence, or with no confidence at all. The agent that is most sure of being correct ‘wins’. For instance, the LOC agent and the TMP agent might each want to assign its tag to a certain constituent; but the TMP agent has a higher confidence for that constituent—it is more sure of being correct—so the constituent gets tagged TMP. If the group of agents as a whole guesses wrong, they adjust their parameters slightly in order to get similar instances correct in the future. To continue the example, if the TMP agent ‘wins’, but the correct answer would have been LOC, then the TMP agent decreases its confidence on that stimulus and similar stimuli, while the LOC agent increases them accordingly.

We can look at this visually as well, though it is a little more complicated than the binary case (and yet much more simplified relative to what’s actually happening). In Figure 6.3, we see a tri-valued dataset. The three solid lines represent the perceptrons—the experts. The distance from a line to a point represents that expert’s confidence in its judgement about that point. Some points are ‘claimed’ by more than one expert. Those points in the upper middle of the figure are claimed both as crosses and as Xs, but those to the left of the dashed line are claimed with more confidence by the cross perceptron, while those to the right of the dashed line are claimed more strongly by the X perceptron.

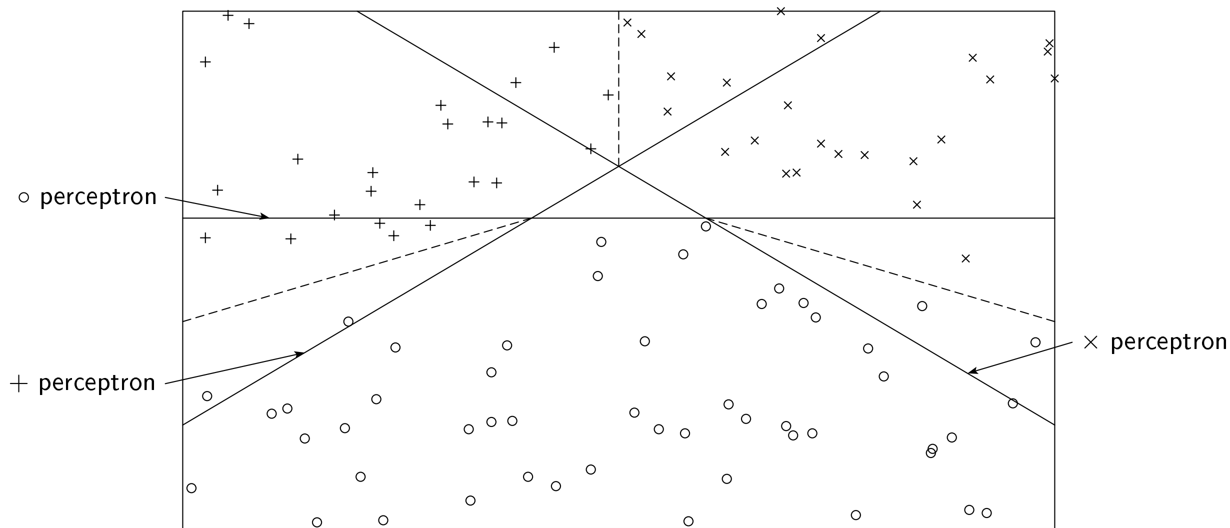


Figure 6.3: A tri-valued dataset

For $j$ in $1 \ldots m$
    Zero weight vector $w_j$
For each constituent vector $c_i$, with correct tag $f$
    For $j$ in $1 \ldots m$
        $s_j \leftarrow w_j \cdot c_i$
    $a \leftarrow \text{argmax}_j\, s_j$
    if $a \neq f$        (*guessed wrong*)
        $w_a \leftarrow w_a - c_i$
        $w_f \leftarrow w_f + c_i$

Figure 6.4: The perceptron algorithm: training

### 6.2.1 The algorithm

What makes the basic perceptron algorithm so simple, and so fast, is that the process of 'consulting' each expert is just a matter of calculating a dot product; and then correcting them when they are wrong is just vector addition. The full algorithm for $m$-valued classification is given in Figure 6.4. Each expert is a weight vector $w_j$, its confidence measured by taking its dot product with the stimulus, constituent $c$. This confidence, or score, is compared against the scores given by the other weight vectors; and the one with the greatest score gets to assign its tag, $a$, to the constituent. If $a$ is the same as the correct tag $f$, we're done. If not, then our system guessed wrong.

When the system guesses wrong, we need to update the weights. This is done by simple vector addition: $c_i$ is subtracted from the weight vector that (incorrectly) scored it highest, and added to the weight vector that *should* have scored it highest. This has the direct effect of nearly guaranteeing that the system will get $c_i$ right if it sees it again—if there are twenty active features, then the score

For each constituent vector $c$
  For $j$ in $1 \ldots m$
    $s_j \leftarrow w_j \cdot c$
  $a \leftarrow \text{argmax}_j\, s_j$
  Mark $c$ with tag $a$

Figure 6.5: The perceptron algorithm: testing

$w_a$ outputs will have decreased by twenty, and the score $w_f$ outputs will have increased by twenty. (Remember that the constituent vector $c_i$ is binary.) But more importantly, this also has the effect that constituents *similar* to $c_i$ will be similarly affected, albeit to a lesser degree.

The testing algorithm is essentially identical to the training algorithm, except that the weights are not zeroed at the beginning and are not updated at each iteration. We give the testing algorithm in Figure 6.5, for completeness.

The basic perceptron algorithm is, if nothing else, *fast*. The training is $O(mfn)$, where $n$ is the number of training examples, $f$ the number of features in use (varying from 10–20 in this work), and $m$ the number of possible values (no more than 10 here). Individual test cases are $O(mf)$, essentially constant. Put into concrete numbers, a full run through the training set (780K constituents) takes about three to five minutes, and the entire test set (27K constituents) about six seconds, of which a significant portion is just the overhead of reading in the files. Even training for twenty 'epochs', or iterations through the full training set, only takes about an hour. Unfortunately, the speed comes at a cost: the algorithm never actually converges, and even after it stabilises, the fluctuations in performance can be on the order of 7–10 percentage points. It is a tolerable quick and dirty solution: very quick, but very dirty.

## 6.3 Voted perceptrons

One proposed solution to the convergence problem is the *voted perceptron* (Freund and Schapire, 1999). The basic idea behind this algorithm is that the wildly fluctuating performance of the basic perceptron is producing a large number of pretty good classifiers; and while any given one of them may be the worst of the lot, if we use them all together to find a result, we should do reasonably well. Expressed in terms of Figure 6.2, the basic perceptron would keep shuffling the line around, so that at any given time the points near the line might be on the wrong side of it. But with the exception of the new point, most of the points would be on the correct side of the line more often than not, and hence the voted perceptron would put them all on the correct side of the line.

The essence of the voted perceptron training algorithm (shown in Figure 6.6) is identical to the basic perceptron algorithm; we merely add a bit of bookkeeping. Specifically, new weight vectors are created rather than updating the old ones—we need to retain all weight vectors for later use in the voting. The new $t_k$ variable represents how long a given weight vector $w_k$ has gone unchanged,

For $j$ in $1 \ldots m$
  Zero weight vector $w_{0,j}$
$k \leftarrow 0$
$t_0 \leftarrow 0$
For each constituent vector $c_i$, with correct tag $f$
  For $j$ in $1 \ldots m$
    $s_j \leftarrow w_{k,j} \cdot c_i$
  $a \leftarrow \text{argmax}_j\, s_j$
  if $a \neq f$ (*guessed wrong*)
    $w_{k+1,a} \leftarrow w_{k,a} - c_i$
    $w_{k+1,f} \leftarrow w_{k,f} + c_i$
    $k \leftarrow k + 1$
    $t_k \leftarrow 1$
  else
    $t_k \leftarrow t_k + 1$

Figure 6.6: The voted perceptron algorithm: training

For each constituent vector $c$
  For $k$ in $1 \ldots M$
    For $j$ in $1 \ldots m$
      $s_{k,j} \leftarrow w_{k,j} \cdot c$
    $a \leftarrow \text{argmax}_j\, s_{k,j}$
    $v_a \leftarrow v_a + t_k$
  $b \leftarrow \text{argmax}_a\, v_a$
  Mark $c$ with tag $b$

Figure 6.7: The voted perceptron algorithm: testing

again for use in the voting later. (We could alternatively omit the $t$ variable, and instead create a new $w_{k+1}$ in *every* iteration, identical to $w_k$ in the cases where we guess correctly.)

The testing section is significantly more complicated in the voted perceptron. There is an extra loop, with $M$ iterations, where $M$ is the number of mistakes made during training (and hence the number of different extant weight vectors). The inner loop is essentially the same as the old basic testing algorithm: calculate scores for each possible output value, and choose the one with the highest score. But here, that value ($a$) just gets a certain number of votes according to how long the current weight vector survived during training ($t_k$). Then at the end, the value with the most votes is selected and returned.

The chief disadvantage of this algorithm is that it is slow. While the training phase is essentially identical, running a single test case is now $O(mfM)$—and $M$ is in the many thousands even for a single epoch of training. For easy tasks like syntactic tagging, where fewer mistakes are made, the testing phase (again, over 27K constituents) can take as much as two hours; the semantic tagging task can take twelve or more. On the other hand, at least the training is speedy; as previously

```
For i in 1...n, j in 1...m
    α_{i,j} ← 0
For each constituent vector c_i, with correct tag f
    For j in 1...m
        s_j ← Σ_{k=0}^{i-1} α_{k,j} c_k · c_i
    a ← argmax_j s_j
    if a ≠ f        (*guessed wrong*)
        α_{i,a} ← −1
        α_{i,f} ← +1
```

Figure 6.8: The dual form of the perceptron algorithm: training

```
For each constituent vector c
    For j in 1...m
        s_j ← Σ_{k=0}^{i-1} α_{k,j} c_k · c_i
    a ← argmax_j s_j
    Mark c with tag a
```

Figure 6.9: The dual form of the perceptron algorithm: testing

stated, the training is essentially the same as before, and can get through an epoch in three to five minutes.

As formalised above, the algorithm has another major disadvantage: it would require large amounts of storage, for all of the weight vectors. Conveniently, though, since all modifications to weight vectors are the addition or subtraction of some constituent vector $c_i$, we can instead just store the index $i$ along with a multiplier of $+1$ or $-1$, and then reconstruct the weight vector on the fly later. This variant on the perceptron algorithm is known as the "dual form", and can be applied without penalty to any of the perceptron variants with a factor of $M$ in their complexity (as these are already iterating through all the historical weight vectors anyway). The dual form algorithm in its non-voted form is given in Figures 6.8 and 6.9. The key thing to note is that the equivalence

$$w_{k,j} \equiv \sum_{k=0}^{i-1} \alpha_{k,j} c_k$$

holds throughout the algorithm.

### 6.3.1 Sparse voted perceptron

A voted perceptron trained on just a single epoch through the training data took over twelve hours to run through the testing data—27K constituents—averaging about a second and a half per constituent. As this is already significantly longer than any real-world algorithm will likely want to wait,

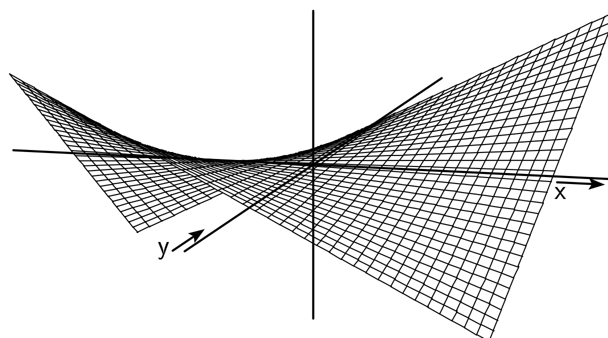


Figure 6.10: The surface $z = xy$

there seemed to be little point in running it on multiple epochs of training (which would increase the time *quadratically*). But the training goes so fast! Isn't there some way to get the advantages of the voted perceptron without having to wait quite so long for it?

So we wrote a 'sparse' voted perceptron. Rather than taking every single weight vector that was created during training, we sample them. That is, during the training process, at various intervals we save the current weights vectors; in the testing, we use just the saved vectors to vote—on the order of dozens, rather than hundreds of thousands. This removes the $M$ factor from the complexity, making the testing $O(mfn)$ again. Now that we (again) had access to many epochs of training, we elected to only save weights from after the first five epochs—allowing the perceptrons to, arguably, converge. With 61 weights vectors voting, the testing run took about four minutes, and performed quite well.

## 6.4 Kernel-based perceptrons

Another tactic used to improve the basic perceptron algorithm is to reduce its reliance on the linear separability of the data. In some cases, the data may be separable by a polynomial, curved separator; even if not, such a separator may at least do a better job. The problem of finding one is solved by reducing it to the previously solved problem: we have an algorithm that finds linear separators, so we create a mathematical context where (some types of) non-linear separators *look like* linear separators, and then let the perceptron discover them.

Consider a binary classification problem that makes use of two features. Each stimulus can be plotted on a two-dimensional $(x, y)$ coordinate plane. Now suppose that we create an additional

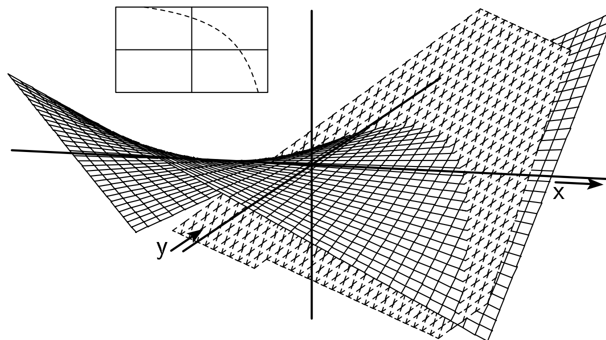


Figure 6.11: A plane in 3D corresponds to a hyperbola in 2D

feature that is nothing more than the product of the existing two features. The stimuli have now been embedded into a three-dimensional $(x, y, z)$ coordinate space, but the three dimensions are not independent, and every stimulus thus lies on the curved surface $z = xy$, shown in Figure 6.10.

But as far as the perceptron algorithm is concerned, a 3D space is a 3D space, and it will go to its business of learning a linear separator—a plane—within that 3D space. However, because we have projected the original dataset onto a curved surface, the only relevant part of that plane with regards to separating the data is the intersection between the plane and the curved surface; and this can be a curved line. Figure 6.11 shows a plane intersecting the curved surface, and the inset graph in that figure shows the projection of that intersection back onto the $(x, y)$ plane: a non-linear separator.

In practice, when the base feature set is already a fairly high-dimensional space (a few lexical features will easily put a binary feature set into the hundreds of thousands of features), it would be prohibitively expensive to generate conjunction features for every pair of them. Hence we need some way of simulating them without actually generating them: enter kernels.

In general, a kernel expresses the similarity between two stimuli $\mathbf{x}$ and $\mathbf{y}$. The kernel function $K(\mathbf{x}, \mathbf{y})$ must be rewritable as a dot product $\Phi(\mathbf{x}) \cdot \Phi(\mathbf{y})$, where $\Phi$ is some function that returns a vector as output. It is used as a replacement for the simple dot product $c_k \cdot c_i$ in the dual-form algorithm (Figures 6.8 and 6.9 on page 39). If $\Phi$ is the identity function, then the kernel algorithm is exactly the dual form algorithm.

As a first pass at a conjunction-based $\Phi$, we might try

$$\Phi(\mathbf{x}) = \langle x_0, x_1, x_2, \ldots, x_f, x_0 x_1, x_0 x_2, \ldots, x_{f-2} x_{f-1}, x_{f-1} x_f \rangle \ ,$$

that is, a list of all the individual features, followed by an all-pairs list of feature products. But of

course this suffers from the problem mentioned above: it is prohibitively expensive when $f$ is likely to be on the order of 100,000. Instead, Freund and Schapire propose the following $\Phi$:

$$\Phi(\mathbf{x}) = \left\langle 1, \sqrt{2}x_0, \sqrt{2}x_1, \ldots, \sqrt{2}x_f, \sqrt{2}x_0x_1, \sqrt{2}x_0x_2, \ldots, \sqrt{2}x_{f-1}x_f, x_0^2, x_1^2, \ldots, x_f^2 \right\rangle .$$

This somewhat more complicated $\Phi$ function yields the following kernel:

$$\begin{aligned} K(\mathbf{x}, \mathbf{y}) &= \Phi(\mathbf{x}) \cdot \Phi(\mathbf{y}) \\ &= 1 + 2x_0y_0 + \cdots + 2x_fy_f + 2x_0x_1y_0y_1 + \cdots + 2x_{f-1}x_fy_{f-1}y_f + x_0^2y_0^2 + \cdots + x_f^2y_f^2 \\ &= (1 + x_0y_0 + x_1y_1 + \cdots + x_fy_f)^2 \\ &= (1 + \mathbf{x} \cdot \mathbf{y})^2 \end{aligned}$$

This can be calculated directly from the feature vectors, with essentially no extra work. More generally, the expression $(1 + \mathbf{x} \cdot \mathbf{y})^d$ is a $d$-dimensional kernel that enables the perceptron to learn polynomial separators of degree $d$.

Unfortunately, a side effect of using the kernel-based perceptron with higher-dimensional kernels is that the dual form must be used even for training; which makes the training algorithm $O(mfMn)$—again, $m$ and $f$ are roughly constant factors, but $Mn$ is essentially an $n^2$ term. Undesirable at best when working with a training set in the 780K range—the datasets on which these techniques were developed tended to fall in the 40–50K range(Freund and Schapire, 1999). The good news is that the performance increase on the higher-dimensional kernels makes these run considerably faster than a $d = 1$ kernel training run, but it is still extremely slow, taking anywhere from a day (on the syntactic tagging problem) to a week (on the semantic tags). Testing time is comparable to that for the voted perceptron, ranging from two to twelve hours.

## 6.5 Feature sets

The feature sets we employed can be divided into three main groups—the minimalist sets, the basic-plus sets, and the full sets. For the minimalist sets, we have just a few features employed: typically the label and head of the constituent under consideration ("self"), plus one or two other features. The features used in the minimalist sets are grouped together into the "basic" set, and then the basic-plus sets are the entire basic set plus one or two extra features. Finally, some runs were performed on sets using all or nearly all of the features available to the system. The exact breakdown is shown in Table 6.2 on page 44. Some abbreviations used in that table are given in Table 6.1 on the next page.

## 6.6 Results

We ran the basic perceptron over a number of different feature sets, attacking both the syntactic and semantic tags. For the larger feature sets (and the harder problems), it sometimes took awhile

| | |
|---|---|
| parlab | parent's label |
| par | parent's label and head |
| siblab | siblings' labels |
| sib | siblings' labels and heads |
| gp | grandparent's label and head |
| ccp | ccparent |
| twosib | labels of siblings at distance 2 |
| psm | parent's semantic tag |
| alt | self's alt head |

Table 6.1: Abbreviations used in feature set names

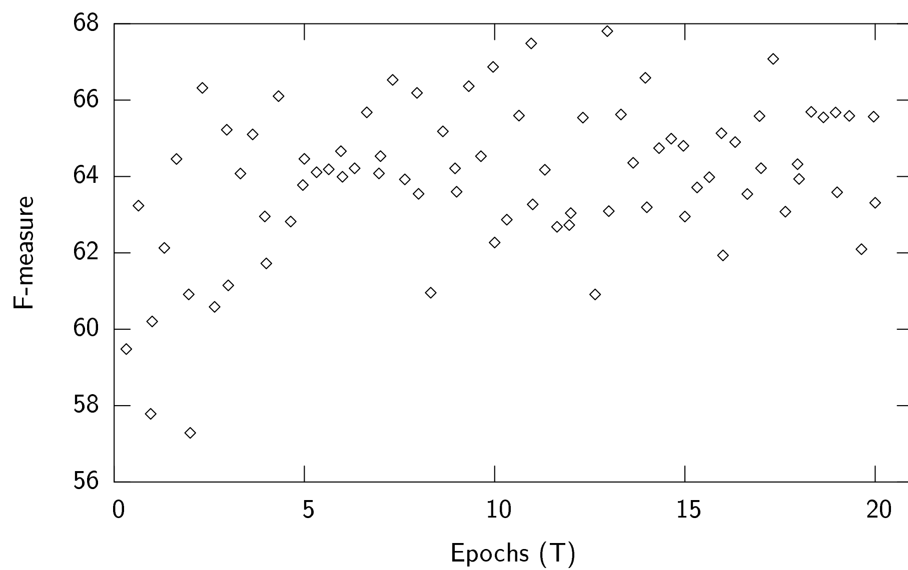


Figure 6.12: Basic perceptron using ecbasic feature set for semantic tags

| | Self | | | | | | | Parent | | | | | | Grandpar | | | | Left sib | | | | Right sib | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | label | head POS | head | alt head POS | alt head | is aux | syn/sem tag | label | cc-label | head POS | head | is aux | sem tag | label | head POS | head | is aux | label | head POS | head | is aux | label | head POS | head | is aux | 2nd-left sib label | 2nd-right sib label |
| self | • | • | • | | | | | | | | | | | | | | | | | | | | | | | | |
| self+parlab | • | • | • | | | | | • | | | | | | | | | | | | | | | | | | | |
| self+par | • | • | • | | | | | • | | • | • | | | | | | | | | | | | | | | | |
| self+siblab | • | • | • | | | | | | | | | | | | | | | • | | | | • | | | | | |
| self+sibs | • | • | • | | | | | | | | | | | | | | | • | • | • | | • | • | • | | | |
| self+gp | • | • | • | | | | | | | | | | | • | • | • | | | | | | | | | | | |
| ecbasic-ccp-gp | • | • | • | | | | | • | | • | • | | | | | | | • | | | | • | | | | | |
| ecbasic-ccp | • | • | • | | | | | • | | • | • | | | • | • | • | | • | | | | • | | | | | |
| ecbasic | • | • | • | | | | | • | • | • | • | | | • | • | • | | • | | | | • | | | | | |
| ecbasic+sm/sy | • | • | • | | | | • | • | • | • | • | | | • | • | • | | • | | | | • | | | | | |
| ecbasic+sibs | • | • | • | | | | | • | • | • | • | | | • | • | • | | • | • | • | • | • | • | • | • | | |
| ecbasic+twosib | • | • | • | | | | | • | • | • | • | | | • | • | • | | • | | | | • | | | | • | • |
| ecbasic+psm | • | • | • | | | | | • | • | • | • | | • | • | • | • | | • | | | | • | | | | | |
| ecbasic+aux | • | • | • | | | • | | • | • | • | • | | | • | • | • | | • | | | | • | | | | | |
| ecbasic+allaux | • | • | • | | | • | | • | • | • | • | • | | • | • | • | • | • | | | • | • | | | • | | |
| ecbasic+alt | • | • | • | • | • | | | • | • | • | • | | | • | • | • | | • | | | | • | | | | | |
| ecbasic+aux+alt | • | • | • | • | • | • | | • | • | • | • | | | • | • | • | | • | | | | • | | | | | |
| fullcorp2-ccp | • | • | • | • | • | • | | • | | • | • | • | • | • | • | • | • | • | • | • | • | • | • | • | • | • | • |
| fullcorp2-lex | • | • | | • | • | • | | • | • | • | | • | • | • | • | | • | • | • | | • | • | • | | • | • | • |
| fullcorp2 | • | • | • | • | • | • | | • | • | • | • | • | • | • | • | • | • | • | • | • | • | • | • | • | • | • | • |
| fullcorp2+sm/sy | • | • | • | • | • | • | • | • | • | • | • | • | • | • | • | • | • | • | • | • | • | • | • | • | • | • | • |

Table 6.2: Features used in each feature set

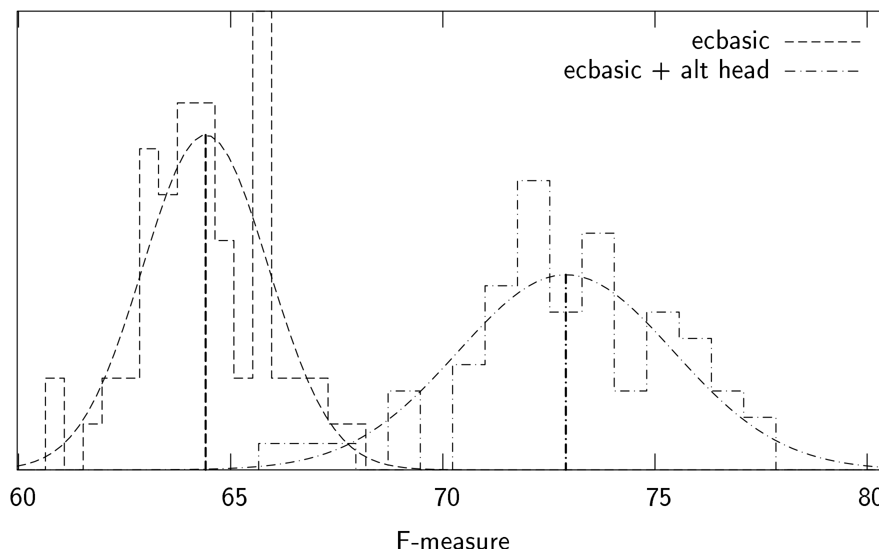


Figure 6.13: Performance of the basic perceptron on the semantic tags, given similar feature sets

for the training to really "take hold", but typically by the end of the fifth epoch, the training had converged as much as it ever would. To measure this, we saved the weights after every 250,000 training constituents (about a third of an epoch), and then re-ran the testing algorithm on the development corpus with each of those weights in turn. A plot of the results of this process on a representative problem is given in Figure 6.12 on page 43. In this graph we can see that the algorithm quickly (before the first test run) learns the easy examples, then takes a few epochs to learn the slightly harder cases, before settling into a six-point fluctuation around 64%.

It is worth noting that the scatter does indeed seem to be normally distributed. We can plot a histogram of the system's performance, and an overlay of the corresponding normal distribution usually seems to be a good fit, as in Figure 6.13. In this particular figure, we can clearly see that despite some overlap, adding the 'alt head' feature is a big win when attacking the semantic tagging problem.

For the first round of tests, we used a variety of different combinations of features in the algorithms that run fast. Specifically, we ran the basic algorithm in Figure 6.4 on page 36 on the training set—sections 2–21 of the WSJ portion of the Penn Treebank II—until we had seen each constituent twenty times, recording the weights at the end of every epoch as well as after 250K, 500K, and 750K constituents of the epoch. We threw out the early weights vectors for each training set, on the basis that the perceptrons had not been fully trained yet; for the threshold we arbitrarily picked the end of the fifth epoch. With the weights vectors that remained, then, we ran the basic testing algorithm (Figure 6.5) on the entire development corpus (section 24). This yielded 61 scores, of which we report the mean precision, recall, and F-measure, and the standard deviation on the F-measure, in Table 6.3 on the next page. We then used these 61 weights vectors in our sparse voted perceptron as described in Section 6.3.1 above, yielding the precision, recall, and F-measure results also reported

| Syntactic tags | | | | | | | | Semantic tags | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| basic | | | | sparse voted | | | | basic | | | | sparse voted | | |
| $\mu_P$ | $\mu_R$ | $\mu_F$ | $\sigma_F$ | P | R | F | | $\mu_P$ | $\mu_R$ | $\mu_F$ | $\sigma_F$ | P | R | F |
| 54.42 | 29.75 | 37.87 | 3.31 | 65.20 | 29.39 | 40.51 | self | 48.05 | 44.69 | 45.71 | 7.28 | 61.69 | 46.34 | 52.93 |
| 89.49 | 85.87 | 87.46 | 2.07 | 96.72 | 85.52 | 90.77 | self+parlab | 54.66 | 54.29 | 53.80 | 3.87 | 70.43 | 54.15 | 61.22 |
| 95.62 | 95.30 | 95.43 | 0.53 | 96.39 | 96.74 | 96.57 | self+par | 64.09 | 62.89 | 63.29 | 1.50 | 71.98 | 65.02 | 68.32 |
| 85.25 | 79.28 | 81.71 | 2.98 | 91.62 | 82.36 | 86.74 | self+siblab | 57.31 | 59.77 | 58.25 | 2.59 | 69.50 | 56.59 | 62.38 |
| 93.12 | 91.77 | 92.29 | 1.29 | 95.56 | 93.44 | 94.49 | self+sibs | 59.83 | 59.68 | 59.11 | 3.24 | 69.82 | 60.49 | 64.82 |
| 76.91 | 80.43 | 78.21 | 2.34 | 86.98 | 79.71 | 83.19 | self+gp | 56.12 | 51.00 | 53.14 | 2.51 | 65.40 | 52.83 | 58.45 |
| 96.44 | 97.10 | 96.75 | 0.46 | 97.54 | 98.21 | 97.87 | ecbasic-ccp-gp | 66.03 | 64.90 | 65.35 | 1.48 | 73.00 | 67.12 | 69.94 |
| 97.52 | 97.50 | 97.50 | 0.49 | 98.60 | 98.50 | 98.55 | ecbasic-ccp | 64.70 | 62.83 | 63.56 | 1.60 | 72.67 | 65.12 | 68.69 |
| 97.37 | 97.71 | 97.52 | 0.52 | 98.44 | 98.53 | 98.48 | ecbasic | 65.44 | 63.21 | 64.07 | 1.36 | 73.34 | 65.37 | 69.13 |
| 98.05 | 97.74 | 97.89 | 0.38 | 98.73 | 98.60 | 98.66 | ecbasic+sm/sy | 66.81 | 63.11 | 64.66 | 1.86 | 73.61 | 65.02 | 69.05 |
| 97.94 | 97.61 | 97.77 | 0.32 | 98.53 | 98.40 | 98.47 | ecbasic+sibs | 66.79 | 61.87 | 63.80 | 1.83 | 73.58 | 64.54 | 68.76 |
| 97.56 | 98.19 | 97.87 | 0.53 | 98.57 | 98.73 | 98.65 | ecbasic+twosib | 65.76 | 63.47 | 64.25 | 2.22 | 73.20 | 67.02 | 69.98 |
| 97.63 | 97.74 | 97.68 | 0.36 | 98.60 | 98.47 | 98.53 | ecbasic+psm | 66.85 | 62.35 | 64.24 | 1.84 | 74.96 | 64.39 | 69.27 |
| 97.82 | 97.32 | 97.55 | 0.48 | 98.60 | 98.50 | 98.55 | ecbasic+aux | 66.49 | 63.51 | 64.67 | 1.82 | 74.26 | 65.85 | 69.80 |
| 97.55 | 97.97 | 97.75 | 0.40 | 98.44 | 98.73 | 98.58 | ecbasic+allaux | 65.78 | 62.26 | 63.59 | 2.00 | 74.38 | 64.44 | 69.05 |
| 98.11 | 97.70 | 97.90 | 0.30 | 98.66 | 98.37 | 98.51 | ecbasic+alt | 76.62 | 68.38 | 72.00 | 2.15 | 83.43 | 72.44 | 77.55 |
| 98.06 | 97.77 | 97.91 | 0.33 | 98.60 | 98.43 | 98.52 | ecbasic+aux+alt | 77.33 | 67.49 | 71.77 | 2.25 | 83.90 | 72.68 | 77.89 |
| 97.84 | 98.36 | 98.10 | 0.30 | 98.70 | 98.76 | 98.73 | fullcorp2-ccp | 75.95 | 70.52 | 72.80 | 2.38 | 82.36 | 75.17 | 78.60 |
| 94.94 | 93.98 | 94.42 | 0.85 | 96.67 | 94.68 | 95.67 | fullcorp2-lex | 50.83 | 38.55 | 41.99 | 6.64 | 63.01 | 40.39 | 49.23 |
| 98.07 | 98.24 | 98.15 | 0.35 | 98.92 | 98.63 | 98.78 | fullcorp2 | 77.08 | 69.61 | 72.92 | 1.92 | 81.76 | 73.71 | 77.53 |
| 98.26 | 98.34 | 98.29 | 0.22 | 98.86 | 98.76 | 98.81 | fullcorp2+sm/sy | 75.97 | 71.49 | 73.44 | 2.07 | 82.06 | 75.17 | 78.46 |

Table 6.3: Results of the basic perceptron and sparse voted perceptron, after 20 epochs of training, tested on section 24

| | Syntactic | | | | | Semantic | | |
|---|---|---|---|---|---|---|---|---|
| P | R | F | time | | time | P | R | F |
| 96.29 | 95.56 | 95.92 | 2h43m | self+par | 11h33m | 73.33 | 59.95 | 65.97 |
| 98.97 | 96.93 | 97.94 | 1h51m | ecbasic | 13h26m | 74.89 | 59.66 | 66.41 |
| 99.01 | 97.49 | 98.24 | 1h54m | ecbasic+alt | 10h47m | 81.63 | 64.59 | 72.11 |
| 99.07 | 97.55 | 98.31 | 2h35m | fullcorp2 | 19h22m | 82.00 | 64.63 | 72.01 |

Table 6.4: Voted perceptron results

| | | nonvoted | | | voted | | |
|---|---|---|---|---|---|---|---|
| | | P | R | F | P | R | F |
| **Syntactic** | | | | | | | |
| self+par | $d = 2$ | 95.26 | 96.3 | 95.8 | 97.1 | 96.0 | 96.5 |
| ecbasic | $d = 2$ | 97.58 | 97.4 | 97.5 | 99.0 | 97.8 | 98.4 |
| ecbasic | $d = 3$ | 97.59 | 97.7 | 97.6 | 98.9 | 98.1 | 98.5 |
| ecbasic | $d = 5$ | 99.0 | 98.4 | 98.7 | 98.9 | 98.1 | 98.5 |
| ecbasic+alt | $d = 2$ | 96.73 | 96.5 | 96.6 | 98.7 | 97.8 | 98.2 |
| fullcorp2 | $d = 2$ | 98.09 | 97.4 | 97.7 | 99.0 | 97.8 | 98.4 |
| **Semantic** | | | | | | | |
| ecbasic | $d = 2$ | 83.5 | 73.2 | 78.0 | 83.7 | 71.9 | 77.3 |

Table 6.5: Kernel perceptron results

in Table 6.3.

Due to the expense, we did not run the true voted perceptron algorithm on all of our feature sets, merely picking a representative few. Runs are based on the alpha values after just one epoch of training. Table 6.4 contains the results of these experiments. The results are lower in almost every case than those for the sparse voted perceptron (henceforth SVP); but this is to be expected, as the SVP had access to fully twenty epochs of training, and indeed completely ignored all the early first-epoch weight vectors, upon which the voted perceptron is relying.

Again with the kernel perceptron, we have chosen just a few feature sets on which to gather statistics. We trained on a single epoch through the training sections, then ran the testing algorithm both with and without voting. The results are reported in Table 6.5. Predictably, the results are much better now that feature conjunctions are in play; but we ran even fewer tests here, especially on the semantic tags, as the training stage was prohibitively long. While normally a lengthy training phase is not a huge cause for concern (since real applications only train once), the training phase on the semantic tags was on the order of *weeks*, making repeated trials difficult at best.

To close out this chapter, we present a breakdown of the results of the SVP run trained on feature set fullcorp2-ccp, according to how the tagger does on each tag, in Table 6.6 on the following page. We did not run sparse voting on any of the kernel-based tests—since the kernel algorithm requires checking each training example anyway, there is no reason not to consider all votes.

| | Syntactic | | | | Semantic | | |
|---|---|---|---|---|---|---|---|
| | P | R | F | | P | R | F |
| DTV | 72.22 | 86.67 | 78.79 | ADV | 77.47 | 80.10 | 78.76 |
| LGS | 88.12 | 90.82 | 89.45 | BNF | 00.00 | 00.00 | 00.00 |
| PRD | 97.68 | 97.20 | 97.44 | DIR | 79.52 | 53.66 | 64.08 |
| PUT | 83.33 | 90.91 | 86.96 | EXT | 79.25 | 80.77 | 80.00 |
| SBJ | 99.70 | 99.74 | 99.72 | LOC | 79.46 | 70.89 | 74.93 |
| SBJ | 99.70 | 99.74 | 99.72 | MNR | 75.61 | 56.36 | 64.58 |
| VOC | 00.00 | 00.00 | 00.00 | NOM | 93.85 | 93.13 | 93.49 |
| | | | | PRP | 68.81 | 71.43 | 70.09 |
| | | | | TMP | 87.50 | 80.62 | 83.92 |

Table 6.6: SVP results with fullcorp2-ccp training, broken down by tag

# Chapter 7

# Analysis

In this chapter we will attempt to interpret the results given earlier in this thesis; we also include a number of results from smaller experiments designed to highlight differences among systems, among feature combinations, or among some other sets of execution parameters.

## 7.1 Feature trees vs. perceptrons

We can compare our different systems on a number of different metrics. The main ones we will specifically consider are training and testing time, and accuracy.

The feature tree system can run its entire training algorithm in just under two and a half minutes on a 1.5GHz Linux box. The fast perceptron training runs through a single epoch in about four or five minutes, handling a twenty-epoch run in about an hour. The slower dual perceptron training algorithm requires a day or two for the 'easy' problems, such as syntactic tagging, and over a week for the 'hard' problems, such as semantic tagging. The training algorithm time is not the most important point of comparison, of course; on a system in actual use, the training need only be run once.

Trading off extremely long training times for shorter test runs makes a great deal of sense in practice, but is not applicable here. In none of the algorithms we tested was there a way to expend more training time to speed up the testing. The 'slow' perceptron algorithms took forever to train, but also forever to test—on the order of an hour or two up to nine or ten hours, just to tag a short, 1400 sentence test corpus. This is simply not a realistic amount of time in which to apply the algorithm. The fast perceptrons fared better—at about six minutes for the test corpus (either easy or hard problems), the tagger approaches real time performance. Feature trees fare better still, requiring just 45 seconds to tag the entire test corpus.

So the feature trees win on speed, with fast perceptrons running a respectable second. What of the accuracy of these systems? Here the results appear to swing the other direction. System performance obviously depends on the features actually used, but even with relatively few features in play, the fast PMV perceptron is competitive with the best feature tree models on the syntactic

tagging task. With all the features available to it, the perceptron model considerably outperforms anything else on this task.

The semantic task does not provide quite such clear-cut results; perhaps unsurprisingly, the perceptron model requires more features to match the feature trees' performance. But in the end, it remains competitive with the feature tree model.

Other, more minor, considerations come into play for the discriminating customer, choosing between these two systems for function tagging. We will consider two in particular. First of all, applications of NLP that require language models have recently been using more complex models than the traditional trigrams; using syntactic structure has become quite popular. It is thus reasonable to hope to integrate function tagging into language modelling. However, this requires each output to have a probability associated to it, or something that can be normalised into a probability: perfect for the feature trees, but the perceptron model is not at all amenable to this.

Secondly, for the user of a function tagger, there are implementation concerns. While neither system is excessively complex, perceptrons are significantly simpler. A researcher who hopes to make use of a function tagger, and already has a separate parser, may find that the extreme ease of coding up the perceptron model may outweigh other concerns. On the other hand, if the parser needs to be coded as well, a combined parser and function tagger can use virtually all of the same underlying feature tree framework, so this may swing the balance over to the feature trees, despite the performance loss in the syntactic tagging. (Realistically, of course, most users of this will probably just find an existing implementation.)

And where do decision trees fit in all of this? Based on the performance of the `c4.5` package, it appears that the training process is much slower than feature trees, if not quite so long as the slow dual perceptron training (used for voted perceptrons and kernel perceptrons). The testing process ranks similarly. Its performance is a slight improvement on the syntactic tags over the feature trees, and a loss on the semantic tags; neither is as good as the best perceptron performance, even restricting view to the fast perceptrons. While not inherent to the model, decision trees often yield probabilistic results, and hence could be used for language modelling. Generally, it is mediocre relative to the others—not crashingly bad, but not an improvement—and its extreme memory requirements make it unsuitable for actual use.

## 7.2 Helpful features

Simple knowledge about the active constituent, the "self", is not sufficient. On syntactic tags, just knowing the self's label and head will grant performance in the 40% range. Knowing just the label of either the parent or the adjacent siblings is crucial; this will improve performance into the 90s—knowing labels of parents and siblings, as well as the parents' head, will push performance nearly as high as it can go. Adding information about the grandparent can give another half a percentage point, but more exotic features did not significantly improve performance at all.

On the semantic tags, the pattern is similar, but not quite the same. The initial bar is a little

bit higher—knowledge about self only will start a system out at 53%—but the end result isn't nearly as high. Again, adding parent and/or sibling information is important, bringing performance into the mid-60s, but the grandparent is *not* helpful here. In fact, adding the grandparent hurts performance slightly; this can be attributed to the fact that it adds noise to the system without bringing any concomitant improvement. Beyond the parents and siblings, the most useful single gain is clearly found by attending to the alt head: any serious work on semantic tagging should include this feature or one like it. Other features gave small but noticeable improvements, of which the largest (and perhaps the most surprising) came from attending to the nonadjacent siblings, which improved performance on the development corpus by over a percentage point. Each of the extra features seems to contribute different information; using them all together does improve the aggregate performance.

Finally, ignoring the lexical features will approximately double the error rate of the system. In the case of syntactic tags, this may be an acceptable loss, as the resulting system still yields nearly 96% accuracy; but this brings semantic tag performance below 50%, clearly not useful for any real application.

## 7.3 Related work

We now return to some of the systems presented in Section 2.2, reporting their results and comparing them to our own to the extent that that is possible.

### 7.3.1 Collins (1997)

The system Collins used to detect complements is a simple lexicalised generative model, and the complement-marking is performed as an intermediate step in parsing. The features used are the parent's label and head, the label of the sibling that contains the parent's head, the current constituent's distance from that sibling, and the subcategorisation frame of that sibling (which has been previously guessed based on the parent's label and head and the label of the head-containing sibling).

Since this is intended primarily as a way to improve performance in the parsing task, Collins does not report the accuracy of the complement-marking, so we are unable to directly compare our work to his. He does note, however, that inserting this step causes a .7% increase in labelled recall and .5% in labelled precision for the parsing task.

### 7.3.2 Brants, Skut, and Krenn (1997)

The work of Brants et al. (1997) is much easier to compare, however. While there is not an exact match between the information recovered by their system and our systems, there is a great deal of overlap; with certain caveats, we are prepared to compare these results with our syntactic tagging results.

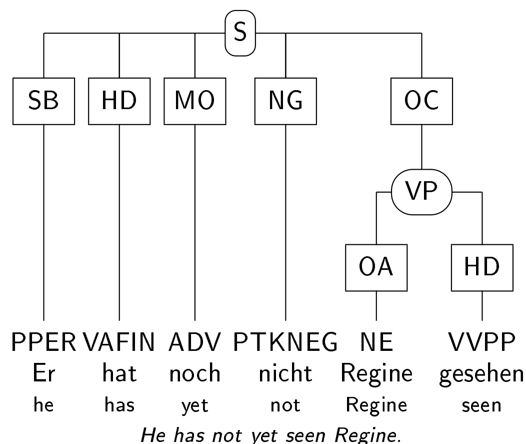


Figure 7.1: A sample sentence with NEGRA-style annotation

Their system works with the NEGRA tagset (see Section 2.2.2), which has many more function tags than the Penn treebank; every constituent has exactly one function tag in this system. Their algorithm relies to some extent on this finer granularity of function tags: they use an order-2 Markov model whose states correspond to the tags themselves. A separate model is trained for each phrasal category, representing the 'parent' of a group of dependencies. The model then moves from state to state (where, again, the states correspond to function tags), emitting the phrasal category labels of the dependent nodes.

For example, on the sentence in Figure 7.1, one model would be trained for S nodes. From the start state, the model moves to the SB ('subject') state with a certain probability, conditioned on the parent (S) and the fact it was the first child. The model then emits the PPER (pronoun) label with a probability based on the parent (S) and the function tag (SB). The model then moves to the HD state, emits VAFIN, and so on. The last state (OC) emits a VP label, and then the model transitions to its stop state with a probability conditioned on the parent (S) and the preceding two function tags (NG, OC). A separate model is consulted to generate the children of the VP node.

Having a function tag on every constituent eliminates any precision-recall tradeoff issues they might have had. It makes their job a bit harder, in that they have more tags (45) to work with; but it also makes their job a bit easier, because many tags will be easy to predict from context—sometimes even deterministically. The Penn treebank, by and large, prefers to leave such constituents untagged.

For instance, we note that their best performance is on tagging the children of PP nodes (on which their accuracy was 97.9%, and which comprise 24% of the data). This is almost certainly because the possible expansions of a PP are relatively constrained—it appears that the vast majority of prepositional phrases in their system will expand to an AC ('adpositional case marker') followed by one or more NKs ('noun kernels'). Comparable phrases in the Penn treebank will have no function tags at all.

A better comparison might be drawn between the F-measure of our syntactic tags (99.0%) and their performance on children of S nodes (89.1%, comprising 26% of the data). Here there are

| **Blaheta** | |
|---|---|
| No-null precision | 96.8% |
| No-null recall | 95.6% |
| No-null F-measure | 96.2% |
| With-null accuracy | 99.1% |
| **Brants, Skut, and Krenn** | |
| PP children | 97.9% |
| S children | 89.1% |
| Overall accuracy | 94.2% |

Table 7.1: A comparison between our work and Brants et al

myriad expansion possibilities, and rather more possible function tags to apply. Four of the five tags shared between the two systems (our SBJ, DTV, PRD, VOC; their SB, DA, PD, VO) will in their system almost always be found on children of S nodes.

Perhaps, in the end, the best comparison would be between their overall accuracy (94.2%) and our 'with-null accuracy' percentage (99.8%), since it takes into account every time we correctly decide to not tag something, which corresponds to a decision in their system to tag a constituent with a non-Penn-treebank tag. On the other hand, this effectively collapses all non-Penn-treebank tags into one tag for us but not for them, which would give us an unfair advantage.

A summary of these results can be seen in Table 7.1.

### 7.3.3 Preiss (2003)

In her 2003 EACL paper, Preiss attempts to compare several different parsers according to the 'grammatical relations' metric (Carroll et al., 1998). In order to evaluate the Charniak (2000) and Collins (1996; 1997) parsers, she manually created a set of deterministic rules for extracting these relations from a Penn treebank-style tree, as output by these parsers. In the context of this thesis, we can examine her accuracy results for the Charniak parser and use them to compare her set of hand-built function tagging rules to our own statistical model.

Again, the tags are not directly comparable, but some have a fairly strong overlap. The ncsubj grammatical relation corresponds fairly closely to the SBJ tag in the Penn set (strictly, it's a subset—csubj, xsubj, and subj are sufficiently rare that we can ignore them for the moment). Preiss reports performance on this tag as 81.80% precision and 70.13% recall. To some extent, this number is low because of parser error—the Charniak parser only performs at 90% or so accuracy—but the numbers are still lower than they would be under our statistical function tagger. Multiplying the parser performance by our own performance on the SBJ tags yields a lower bound of 88.1% precision and 87.1% recall for this task. This estimate would go up if we factored in that the Charniak parser probably does better than average on 'easy' tasks like the subject nodes (as opposed to, say, coordination constructions); and the fact that we may in some cases correctly label SBJ constituents even where the Charniak parser is incorrect.

| | Preiss | | Blaheta (lower bound) | |
|---|---|---|---|---|
| | P | R | P | R |
| ncsubj/SBJ | 81.8 | 70.1 | 88.1 | 87.1 |
| ncmod/sem. tags | 79.8 | 46.3 | 82.1 | 72.1 |

Table 7.2: Comparison against Preiss hand-built tagger

A similar lower bound can be derived for the ncmod tag, which corresponds approximately to all semantic tags other than NOM; here Preiss reports 79.84% precision and just 46.32% recall, but an estimate against our semantic tag performance (ignoring mistags) gives 82.1% precision and 72.1% recall. Mistags are ignored because ncmod is not sensitive to distinctions between our semantic tags—hence a mistag, such as applying TMP instead of LOC, should count as correct for the purposes of ncmod. This is still a lower bound, for all the reasons given in the previous paragraph. This Preiss comparison is summarised in Table 7.2.

### 7.3.4 Gildea and Jurafsky (2002)

Gildea and Jurafsky (2002) developed a system to recover the frame element annotation from the FrameNet corpus. (See Section 2.2.4.) As with our system, they make use of syntactic structure for this task; the system is a straightforward empirically-trained statistical model, with each guess conditioned on a small number of features.

The features they used were the phrase label, head word, 'governing category', 'position', 'voice', and 'parse tree path'. The first two are essentially as defined in our system. The governing category is similar to a parent label; but it is only defined for NP constituents, and its value is always S or VP—if the parent is neither of those, a search commences up the tree until one is found. This search is meant to counteract parser attachment error and coordinating constructions; it is not clear how PP nodes play out for this label. The position is binary, and simply indicates whether a constituent is to the left or right of its governing predicate. Voice indicates whether a verb (presumably again the governing predicate) is active or passive. Finally, a parse tree path records the path through the parse tree from the governing predicate to the current constituent—it is a string containing the label of each node along that path, and an indication for each edge whether the path is travelling up the tree or down. (A sample path given for a sentence subject is "VB↑VP↑S↓NP".)

The actual system calculates eight conditional distributions, of varying levels of specificity, according to empirical observations. These eight distributions are interpolated to calculate the final distribution; the system's performance is then 78.5%, assuming that the locations of the frame elements are given as input to the algorithm. A separate algorithm to first find the frame element boundaries scores 66% perfect accuracy plus another 15% partially correct. A system that combines the two tasks—and is thus more comparable to our own—achieves 64.6% precision and 61.2% recall. (Note that these numbers do account for parser error as well—a misparsed frame element is a guaranteed error for the system, yielding a theoretical maximum around 90%.)

One of the main problems in comparing our work to Gildea and Jurafsky has been that with

| Role/Tag | Gildea & Jurafsky P (Known) | R (Unknown) | Blaheta (lower bound) P (Unknown) | R (Unknown) |
|---|---|---|---|---|
| Agent | 92.8 | 76.7 | | |
| SBJ | | | 88.02 | 86.90 |
| LGS | | | 79.98 | 79.89 |
| Manner/MNR | 70.4 | 48.6 | 63.83 | 61.75 |
| Location/LOC | 63.3 | 47.6 | 79.65 | 74.40 |
| Total | 82.1 | 63.6 | | |
| Syntactic | | | 86.43 | 85.32 |
| Semantic | | | 77.67 | 71.85 |

Table 7.3: Some results from Gildea and Jurafsky (2002)

so many more possible tags, the difficulty of the task was hard to estimate. In their 2002 journal article, however, they publish the results of an experiment where they map all the frame element tags into a set of 18 roles, which they used to train a system and then test; this make the system more like the other function tagging tasks we have seen. Several of these are comparable to some of the Penn function tags, and some of them are reported in Table 7.3. In this table, their precision is based on a run where the frame element boundaries were known, but the recall is based on a run where the frame element boundaries were not known in advance. For our own results, it was not known in advance which constituents had function tags, but those elements that were misparsed were originally discarded from the evaluation—to make the comparison more fair, we have again reported our score as a lower bound, calculated by multiplying our own performance by the parser accuracy of 89.5%. This is almost certainly significantly lower than the actual performance on these tasks. A further consideration is that their tags are only placed on complements, while the majority of the Penn semantic tags (e.g. MNR, LOC) are on adjuncts—not subcategorised by the verb, and therefore harder to guess. Even so, we do very well on recall and comparably on precision.

## 7.4 Error analysis and fixing the treebank

In the course of our research, we wanted to attain some understanding of what sort of errors the system was making. While still working primarily with the feature tree system, we took the output of the system and examined each error, determining whether the error was in the algorithm or in

| | |
|---|---|
| Algorithm error | 44% |
| Parse error | 20% |
| Treebank error | 18% |
| Type C error | 13% |
| Dubious | 6% |

Table 7.4: Analysis of reported errors

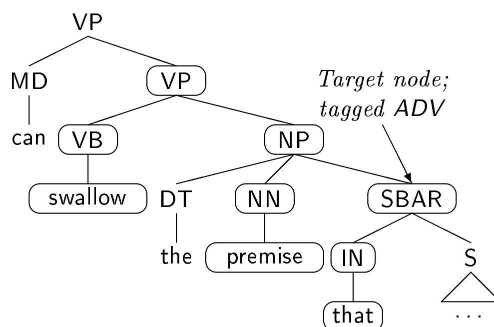


Figure 7.2: SBAR and conditioning info

the treebank, or elsewhere, as reported in Table 7.4. Of the errors we examined, less than half were due solely to an algorithmic failure in the function tagger itself. The next largest category was parse error: this function tagging algorithm requires parsed input, and in these cases, that input was incorrect and led the function tagger astray; had the tagger received the treebank parse, it would have given correct output. In just under a fifth of the reported "errors", the algorithm was correct and the treebank was definitely wrong. The remainder of cases we have identified either as Type C errors—wherein the tagger agreed with many training examples, but the "correct" tag agreed with many others (see Section 7.6.3)—or at least "dubious", in the cases that weren't common enough to be systematic inconsistencies but where the guidelines did not clearly prefer the treebank tag over the tagger output, or vice versa.

## 7.5 Parse error

It may at first seem strange that so many reported errors would be due to parse error, since (as stated in Section 2.4) we only count correctly parsed constituents in our evaluation. However, although the constituent itself may be correctly bracketed and labelled, its exterior conditioning information can still be incorrect. An example of this that actually occurred in the development corpus (again, Section 24 of the treebank) is the 'that' clause in the phrase "can swallow the premise that the rewards for such ineptitude are six-figure salaries", correctly diagrammed in Figure 7.2. The function tagger gave this SBAR an ADV tag, indicating an unspecified adverbial function. This seems extremely odd, given that its conditioning information (nodes circled in the figure) clearly show that it is part of an NP, and hence probably modifies the preceding NN. Indeed, the statistics give the probability of an ADV tag in this conditioning environment as vanishingly small.

However, this was not the conditioning information that the tagger received. The parser had instead decided on the (incorrect) parse in Figure 7.3 on the facing page. As such, the tagger's decision makes much more sense, since an SBAR under two VPs whose heads are VB and MD is indeed rather likely to be an ADV. (For instance, the 'although' clause of the sentence "he can help, although he doesn't want to." has exactly the conditioning environment given in Figure 7.3, except

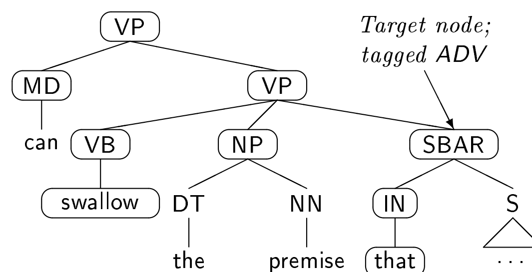


Figure 7.3: SBAR and conditioning info, as parsed

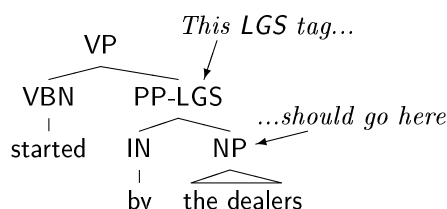


Figure 7.4: A function tag error of Type A

that its left sibling is a comma; and such an SBAR would be correctly tagged ADV.) The SBAR itself is correctly bracketed and labelled, so it still gets counted in the statistics. In fact, since the only requirement for an individual constituent to be "correct" is that it have the correct label and subtend the correct portion of the sentence, every single other conditioning event could (in theory) be wrong! Actual occurrences aren't so extreme, but are sufficiently frequent that they are a significant source of error.

## 7.6 Treebank error

Another thing that lowers the overall performance somewhat is the existence of error and inconsistency in the treebank tagging. Performed by humans, the annotation inevitably has errors in it. We have designed a taxonomy for cataloguing corpus errors and some guidelines for correcting them, first published in (Blaheta, 2002).

### 7.6.1 Type A: Detectable errors

The easiest errors, which we have dubbed "Type A", are those that can be automatically detected and fixed. These typically occur when there would be multiple reasonable ways of tagging a certain interesting situation: the markup guidelines arbitrarily choose one, and the human annotator unthinkingly uses the other.

The canonical example of this sort of thing is the treebank's LGS tag, representing the "logical subject" of a passive construction. It makes a great deal of sense to put this tag on the NP object

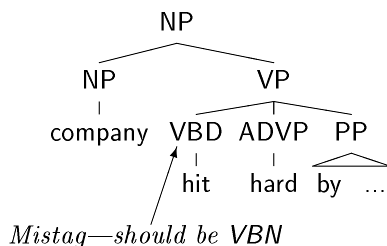


Figure 7.5: A part-of-speech error of Type $B_1$

of the 'by' construction; it makes almost as much sense to tag the PP itself, especially since (given a choice) most other function tags are put there. The treebank guidelines specifically choose the former: "It attaches to the NP object of *by* and not to the PP node itself." (Bies et al., 1995) Nevertheless, in several cases the annotators put the tag on the PP, as shown in Figure 7.4. We can automatically correct this error by algorithmically removing the LGS tag from any such PP and adding it to the object thereof.

The unifying feature of all Type A errors is that the annotator's *intent* is still clear. In the LGS case, the annotator managed to clearly indicate the presence of a passive construction and its logical subject. Since the transformation from what was marked to what ought to have been marked is straightforward and algorithmic, we can easily apply this correction to all data.

### 7.6.2 Type B: Fixable errors

Next, we come to the Type B errors, those which are fixable but require human intervention at some point in the process. In theory, this category could include errors that could be found automatically but require a human to fix; this doesn't happen in practice, because if an error is sufficiently systematic that an algorithm can detect it and be certain that it is in fact an error, it can usually be corrected with certainty as well. In practice, the instances of this class of error are all cases where the computer can't detect the error for certain. However, for all Type B errors, once detected, the correction that needs to be made is clear, at least to a human observer with access to the annotation guidelines.

Certain Type B errors are moderately easy to find. When annotators misunderstand a complicated markup guideline, they mismark in a somewhat predictable way. While not being totally systematically detectable, an algorithm can leverage these patterns to extract a list of tags or parses that *might* be incorrect, which a human can then examine. Some errors of this type (henceforth "Type $B_1$") include:

- VBD / VBN. Often the past tense form of a verb (VBD) and its past participle (VBN) have the same form, and thus annotators sometimes mistake one for the other, as in Figure 7.5. Some such cases are not detectable, which is why this is not Type A.[1]

[1]There is a *subclass* of this error which is Type A: when we find a VBD whose grandparent is a VP headed by a

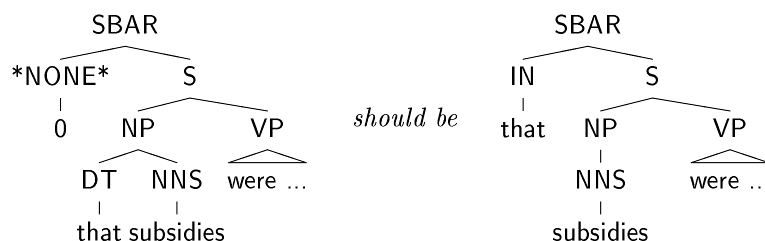


Figure 7.6: A parse error of Type $B_2$

- IN / RB / RP. There are specific tests and guidelines for telling these three things apart, but frequently a preposition (IN) is marked when an adverb (RB) or particle (PRT) would be more appropriate. If an IN is occurring somewhere other than under a PP, it is likely to be a mistag.

Occasionally, an extracted list of maybe-errors will be "perfect", containing only instances that are actually corpus errors. This happens when the pattern is a very good heuristic, though not *necessarily* valid (which is why the errors are Type $B_1$, and not Type A). When filing corrections for these, it is still best to annotate them individually, as the corrections may later be applied to an expanded or modified data set, for which the heuristic would no longer be perfect.

Other fixable errors are pretty much isolated. Within section 24 of the treebank, for instance, we have:

- the word 'long' tagged as an adjective (JJ) when clearly used as a verb (VB)
- the word 'that' parsed into a noun phrase instead of heading a subordinate clause, as in Figure 7.6
- a phrase headed by 'about', as in 'think about', tagged as a location (LOC)

These isolated errors (resulting, presumably, from a typo or a moment of inattention on the part of the annotator) are not in any way predictable, and can be found essentially only by examining the output of one's algorithm, analysing the "errors", and noticing that the treebank was incorrect, rather than (or in addition to) the algorithm. We will call these Type $B_2$.

### 7.6.3 Type C: Systematic inconsistency

Sometimes, there is a construction that the markup guidelines writers didn't think about, didn't write up, or weren't clear about. In these cases, annotators are left to rely on their own separate intuitions. This leaves us with markup that is inconsistent and therefore clearly partially in error, but with no obvious correction. There is really very little to be done about these, aside from noting them and perhaps controlling for them in the evaluation.

Some Type C errors in the treebank include:

form of 'have', we can deterministically retag it as VBN.

- 'ago'. English's sole postposition seems to have given annotators some difficulty. Lacking a postposition tag, many tagged such occurrences of 'ago' as a preposition (IN); others used the adverb tag (RB) exclusively.[2] Since some occurrences really are adverbs, this just makes a big mess.

- ADVP-MNR. The MNR tag is meant to be applied to constituents denoting manner or instrument. Some annotators (but not all) seemed to decide that any adverbial phrase (ADVP) headed by an '-ly' word must get a MNR tag, applying it to words like 'suddenly', 'significantly', and 'clearly'.

The hallmark of a Type C error is that even what *ought* to be correct isn't always clear, and as a result, any plan to correct a group of Type C errors will have to first include discussion on what the correct markup guideline should be.

### 7.6.4 tsed

In order to effect these changes in some communicable way, we have implemented a program called `tsed`, by analogy with and inspired by the already prevalent `tgrep` search program.[3] It takes a search pattern and a replacement pattern, and after finding the constituent(s) that match the search pattern, modifies them and prints the result. For those already familiar with `tgrep` search syntax, this should be moderately intuitive. The program is similar in spirit to `patch`, another standard utility; but where `patch` works on a line level, `tsed` works on a more abstract tree level. Using `patch` for our purposes would only work on a single version of a treebank; even a differently pretty-printed version could not be properly patched, and writing mergeable patches for versions of the treebank that may or may not have been previously patched would be impossible. This shortcoming is overcome with `tsed`.

To the basic pattern-matching syntax of `tgrep`, we have added a few extra restriction patterns (for specifying sentence number and head word), as well as a way of marking nodes for later reference in the replacement pattern (by simply wrapping a constituent in square brackets instead of parentheses).

The replacement syntax is somewhat more complicated, because wherever possible we want to be able to construct the new trees by reference to the old tree, in order to preserve modifiers and structure we may not know about when we write the pattern. For full details of the program's abilities, consult the program documentation, but here are the main ones:

- Relabelling. Constituents can be relabelled with no change to any of their modifiers or children.

- Tagging. A tag can be added to or removed from a constituent, without changing any modifiers or children.

[2] In particular, the annotators of sections 05, 09, 12, 17, 20, and 24 used IN sometimes, while the others tagged all occurrences of 'ago' as adverbs.

[3] `tgrep` was written by Richard Pito of the University of Pennsylvania, and comes with the treebank.

- Reference. Constituents in the search pattern can be included by reference in the replacement pattern.
- Construction. New structure can be built by specifying it in the usual S-expression format, e.g. `(NP (NN snork))`. Usually used in combination with Reference patterns.

Along with `tsed` itself, we distribute a Perl program `wsjsed` to process treebank change scripts like the following:

```
{2429#0-b}<<EOF
NP $ [ADJP] > (VP / keep)        (S \0 \1)
NP <<, markets                   - SBJ
EOF
```

This script would make a batch modification to the zeroth sentence of the 29th file in section 24. The batch includes two corrections: the first matches a noun phrase (NP) whose sister is an ADJP and whose parent is a VP headed by the word 'keep'. The matched NP node is replaced by a (created) S node whose children will be that very NP and its sister ADJP. The second correction then finds an NP that ends in the word 'markets' and marks it with the SBJ function tag.

Distributing changes in this form is important for two reasons. First of all, by giving changes in their minimal, most general forms, they are small and easy to transmit, and easy to merge. Perhaps more importantly, since corpora are usually copyrighted and can only be used by paying a fee to the controlling body (usually LDC or ELDA), we need a way to distribute only the changes, in a form that is useless without having bought the original corpus. Scripts for `tsed`, or for `wsjsed`, serve this purpose.

These programs are available from our website.[4] More complete documentation on `tsed` can be found in Appendix B.

### 7.6.5 Training data

In virtually all empirical NLP work, the training set is going to encompass the vast majority of the data. As such, it is usually impractical for a human (or even a whole lab of humans) to sit down and revise the training. Type A errors can be corrected easily enough, as can some Type $B_1$ errors whose heuristics have a high yield. Purely on grounds of practicality, though, it would be difficult to effect significant correction on a training set of any significant size (such as for the treebank).

Practicality aside, correcting the training set is a bad idea anyway. After expending an enormous effort to perfect one training set, the net result is just one correct training set. While it might make certain things easier and probably will improve the results of most algorithms, those improved results will not be valid for those same algorithms trained on other, non-perfect data; the vast majority of corpora will still be noisy. If a *user* of an algorithm, e.g. an application developer, chooses to perfect a training set to improve the results, that would be helpful, but it is important that researchers report

[4] `http://www.cs.brown.edu/~dpb/tsed/`

results that are likely to be applicable more generally, to more than one training set. Furthermore, robustness to errors in the training, via smoothing or some other mechanism, will also make an algorithm robust to sparse data (that ever-present spectre that haunts nearly every problem in the field); thus eliminating all errors in the training ought not to have as much of an effect on a strong algorithm.

### 7.6.6 Testing data

Testing data is another story, however. In terms of practicality, it is more feasible, as the test set is usually at least one or two orders of magnitude smaller than the training. More important, though, is the issue of fairness. We need to continue using noisy training data in order to better model real-world use, but it is unfair and unreasonable to have noise in the gold standard, which causes an algorithm to be penalised where it is more correct than the human annotation.

As performance on various tasks improves, it becomes ever more important to be able to correct the testing data. A 'mere' 1% improvement on a result of 75% is not impressive, as it represents just a 4% reduction in apparent error, but the same 1% improvement on a result of 95% represents a 20% reduction in apparent error! In the end, a noisy gold standard sets an upper bound of less than 100% on performance, which is if nothing else counterintuitive.

### 7.6.7 Ethical considerations

Of course, we cannot simply go about changing the corpus willy-nilly. We refer the reader to Chapter 7 of David Magerman's thesis (1994) for a cogent discussion of why changing either the training or the testing data is a bad idea. However, we believe that there are now some changed circumstances that warrant a modification of this ethical dictum.

First, we are not allowed to look at testing data. How to correct it, then? An initial reaction might be to "promise" to forget everything seen while correcting the test corpus; this is not reasonable.

Another solution exists, however, which is nearly as good and doesn't raise any ethical questions. Many research groups already use yet another section, separate from both the training and testing, as a sort of development corpus.[5] When developing an algorithm, we must look at some output for debugging, preliminary evaluation, and parameter estimation; so this development section is used for testing until a piece of work is ready for publication, at which point the "true" test set is used. Since we are all reading this development output already anyway, there is no harm in reading it to perform corrections thereon. In publication, then, one can publish the results of an algorithm on both the unaltered and corrected versions of the development section, in addition to the results on the unaltered test section. We can then presume that a corrected version of the test corpus would result in a perceived error reduction comparable to that on the development corpus.

Another problem mentioned in that chapter is of a researcher quietly correcting a test corpus, and publishing results on the modified data (without even noting that it was modified). The solution

[5]In the treebank, this is often section 24.

to this is simple: any results on modified data will need to acknowledge that the data is modified (to be honest), and those modifications need to be made public (to facilitate comparisons by later researchers). For Type A errors fixed by a simple rule, it may be reasonable to publish them directly in the paper that gives the results.[6] For Type B errors, it would be more reasonable to simply publish them on a website, since there are bound to be a large number of them.[7]

Finally, we would like to note that one of the reasons Magerman was ready to dismiss error in the testing was that the test data had "a consistency rate much higher than the accuracy rate of state-of-the-art parsers". This is no longer true.

### 7.6.8 Practical considerations

As multiple researchers each begin to impose their own corrections, there are several new issues that will come up. First of all, even should everyone publish their own corrections, and post comparisons to previous researchers' corrected results, there is some danger that a variety of different correction sets will exist concurrently. To some extent this can be mitigated if each researcher posted both their own corrections by themselves, and a full list of all corrections they used (including their own). Even so, from time to time these varied correction sets will need to be collected and merged for the whole community to use.

More difficult to deal with is the fact that, inevitably, there will be disputes as to what is correct. Sometimes these will be between the treebank version and a proposed correction; there will probably also be cases where multiple competing corrections are suggested. There really is no good systematic policy for dealing with this. Disputes will have to be handled on a case-by-case basis, and researchers should probably note any disputes to their corrections that they know of when publishing results, but beyond that it will have to be up to each researcher's personal sense of ethics.

In all cases, a search-and-replace pattern should be made as general as possible (without being too general, of course), so that it interacts well with other modifications. Various researchers are already working with (deterministically) different versions of corpora—with new tags added, or empty nodes removed, or some tags collapsed, for instance, not to mention other corrections already performed—and it would be a bad idea to distribute corrections that are specific to one version of these. When in doubt, one should favour the original form of the corpus, naturally.

The final issue is not a practical problem, but an observation: once a researcher publishes a correction set, any further corrections by other researchers are likely to decrease the results of the first researcher's algorithm, at least somewhat. This is due to the fact that that researcher is usually not going to notice corpus errors when the algorithm errs in the same way. This unfortunate consequence is inevitable, and hopefully will prove minor.

[6] The rule we used to fix the LGS problem noted in Section 7.6.1 is as follows:

```
{24*-bg}«EOF
NP !- LGS > (PP - LGS)   - LGS
PP - LGS                 ! LGS
EOF
```

[7] The 235 corrections we made to section 24 can be found on our website and in Appendix C.

| **Grammatical** | Precision | Recall | F-measure |
|---|---|---|---|
| Treebank | 96.227% | 94.897% | 95.558% |
| Fixed | 97.018% | 95.137% | 96.068% |
| False error | 20.965% | 4.703% | 11.481% |

| **Form/function** | Precision | Recall | F-measure |
|---|---|---|---|
| Treebank | 81.224% | 76.196% | 78.630% |
| Fixed | 86.743% | 78.078% | 82.182% |
| False error | 29.394% | 7.906% | 16.621% |

| **Topicalisation** | Precision | Recall | F-measure |
|---|---|---|---|
| Treebank | 97.115% | 93.519% | 95.283% |
| Fixed | 99.048% | 94.545% | 96.744% |
| False error | 67.002% | 15.831% | 30.973% |

| **Miscellaneous** | Precision | Recall | F-measure |
|---|---|---|---|
| Treebank | 54.545% | 25.000% | 34.286% |
| Fixed | 66.667% | 30.769% | 42.105% |
| False error | 26.668% | 7.692% | 11.899% |

Table 7.5: Function tagging results, adjusted for treebank error

### 7.6.9 Experimental results

We compiled all the noted treebank errors and their corrections. The most common correction involved simply adding, removing, or changing a function tag to what the algorithm output (with a net effect of improving our score). However, it should be noted that when classifying reported errors, we examined their contexts, and in so doing discovered other sorts of treebank error. Mistags and misparses did not directly affect us; some function tag corrections actually decreased our score. All corrections were applied anyway, in the hope of cleaner evaluations for future researchers. In total, we made 235 corrections, including about 130 simple retags.

Finally, we re-evaluated the algorithm's output on the corrected development corpus. Table 7.5 shows the resulting improvements. Precision, recall, and F-measure are calculated as in (Blaheta and Charniak, 2000). The false error rate is simply the percent by which the error is reduced; in terms of the performance on the treebank version ($t$) and the fixed version ($f$),

$$\text{False error} = \frac{f - t}{1.0 - t} \times 100\%$$

This is the percentage of the reported errors that are due to treebank error.

The main point to be made here is that the false error rates are much higher for precision than for recall, indicating that the main source of treebank error (at least in the realm of function tagging) is due to human annotators forgetting a tag.

### 7.6.10 Further notes on error correction

In this section, we have given a new characterisation of the sorts of noise one finds in empirical NLP, and a roadmap for dealing with it in the future. For many of the problems in the field, the state of the art is now sufficiently advanced that evaluation error is becoming a significant factor in reported results; we show that it is correctable within the constraints of practicality and ethics.

Although our examples all came from the Penn treebank, the taxonomy presented is applicable to any corpus annotation project. As long as there are typographical errors, there will be Type B errors; and unclear or counterintuitive guidelines will forever engender Type A and Type C errors. Furthermore, we expect that the experimental improvement shown in Section 7.6.9 will be reflected in projects on other annotated corpora—perhaps to a lesser or greater degree, depending on the difficulty of the annotation task and the prior performance of the computer system.

An effect of the continuing improvement of the state of the art is that researchers will begin (or have begun) concentrating on specific subproblems, and will naturally report results on those subproblems. These subproblems are likely to involve the complicated cases, which are presumably also more subject to annotator error, and are certain to involve smaller test sets, thus increasing the performance effect of each individual misannotation. As the sizes of the subproblems decrease and their complexity increases, the ability to correct the evaluation corpus will become increasingly important.

# Chapter 8

# Conclusion and future work

The task of function tagging is one for which we are only beginning to discover all the applications. When the annotators of the Penn Treebank added function tags in the early '90s, they were perhaps the first to put this sort of annotation in a significant text corpus, and took on faith that they would prove useful to someone. Projects like FrameNet, the NEGRA corpus, and the Prague Dependency Treebank likewise provided the information thinking it would be useful.

We are now beginning to see researchers explore the range of that utility. Michael Collins tried in 1997 to use them in a limited way to represent the adjunct/complement distinction, to assist in the parsing process; more recently, Julia Hockenmaier has used them in the same way to convert the treebank into a Categorial Grammar formalism. But the true applications are still under development. Just in the few months before the completion of this thesis, researchers from four different universities have expressed interest in using output from the function tagger as input to their own systems, in question answering, information retrieval, and various forms of machine translation. Experiments are still under way, but all were very optimistic that this tool would prove helpful at making their own systems more accurate.

## 8.1 Contributions

The text of this thesis includes, in addition to my own work, a great deal of background and explanations of systems developed by others. Here, then, is a brief summary of the work that is specifically my own.

**Penn function tagger.** This is the first system to specifically work with the function tags used in the Penn treebank, and the first to add function tags to text parsed using the Penn treebank parse structure, phrase labels, and part-of-speech tags. For systems already making use of other Penn-style data markup, it is important to be able to use function tags that are compatible.

**More accurate syntactic tags.** I am not the first to apply syntactic function tags to text, but my system for doing so seems to be considerably more accurate than the existing systems.

**Semantic tags.** I know of no other existing systems that perform a function tag annotation comparable to my semantic category.

**System comparison.** I performed a comparison of several systems on the function tagging task, evaluating the relative advantages of each in various situations.

**New features and feature comparison.** I devised a few new features (most importantly the alt head) that have not been used in statistical systems before, and analysed which features were most helpful in calculating function tags of various categories.

**Voted perceptron modifications.** I created two modifications to the voted perceptron—sampling the weights vectors, and ignoring weights from the first few epochs—that made the technique feasible and improved its performance, respectively.

## 8.2 Future work

One major area of experimentation that I would still like to try is extending this work to other corpora. The NEGRA corpus in particular has available a view of the annotation similar to that used by the Penn treebank, so much of the existing code should be reusable with some slight adaptations. The PDT would also be an interesting corpus to run; in both cases, I suspect that performance would be relatively similar to that seen on the Penn treebank, although NEGRA's tagset is largely restricted to the syntactic domain.

As discussed in Section 7.3.3, the 'grammatical relations' metric has not really been successfully applied to the combination of the Charniak parser and my function tagger. I would like to try this, as I suspect that my system would compare favourably to the other systems presented in Preiss' paper.

Seeing how this work improves other applications remains an important testing ground. As mentioned earlier, other researchers have already requested copies of the system to make use of the function tag information in natural language applications including language modelling, question answering, and machine translation. I eagerly await their results and am interested in doing some work myself in some of these areas. I am confident that my work in function tagging will continue to be a valuable and productive contribution to the field.

# Appendix A

# Function tag descriptions

For a more complete description of each function tag, with more comprehensive examples, see Section 2.2 of (Bies et al., 1995).

## A.1 Syntactic tags

These six tags serve primarily to mark the syntactic role a given constituent plays in a sentence. Verb phrases are assumed to be predicates, and are therefore unmarked. Noun phrases at the sentence or verb phrase level that have neither grammatical nor form/function tags are objects (either direct or indirect, but see A.1.1).

These tags correspond to those presented in Section 2.2.2 'Grammatical role' in (Bies et al., 1995), except that that section also includes topicalisation, which we have separated into a separated category (see A.3).

### A.1.1 DTV—Dative

This tag marks indirect objects in their prepositional form. For this tag to occur, the verb needs to allow the indirect object to appear either before the direct object ('shifted') or in a 'to'-phrase after the direct object ('unshifted'); and the unshifted form must be the one used. (Shifted datives are unmarked.) In the sentence "Alex gave Sasha a book", 'Sasha' is the indirect object but does *not* get a DTV tag; however, in the shifted version "Alex gave a book to Sasha", the phrase "to Sasha" is marked DTV.

If the preposition used is 'for', and the object appears in either shifted or unshifted form, the BNF tag is used (see A.2.2).

### A.1.2 LGS—Logical subject

If a sentence is passive, then the subject of the sentence is what would be the object in an active version of the sentence. The subject of the active version, if it's present at all, is tucked inside a

'by'-phrase under the verb. This "logical" subject is then marked LGS. In the sentence "Alex was hit by Sasha", 'Sasha' would be marked LGS.

It is important to note that the tagging guidelines specifically state that the LGS tag attaches to the NP object of 'by', and not to the PP itself; it is also important to note that the human annotators screwed this up with some regularity (see Section A.1.2).

### A.1.3 PRD—Predicative

Any predicative construction that is not a VP is marked with this tag. For instance (and most commonly), this includes noun and adjective phrase objects of 'be'. In most cases where the PRD-marked phrase is not itself an object of a stative verb like 'be', it heads a VP-less S node, as in "Alex kept the party a secret", where "the party" is the subject and "a secret" the predicate of the S node object of the VP headed by 'kept'.

### A.1.4 PUT—Locative complement of 'put'

This (relatively rare) tag marks the locative complement of 'put'. Since 'put' in its most basic meaning requires more than just a direct object (cf "*Sasha put the book."), but the required location is not a direct/indirect object, it needs to be marked. Thus in "Sasha put the book on the web", "on the web" is marked PUT.

Note, however, that other verbs that behave similarly (e.g. 'set': "*Sasha set the book") do not get this tag.

### A.1.5 SBJ—Subject

The subject of every S node gets this tag. As nearly every "sentence" in the treebank contains one S (or SINV, etc) at the top and often a few nested, this is by far the most common function tag in the treebank.

### A.1.6 VOC—Vocative

This tag is marked on phrases of address; in "Sasha, read the book", 'Sasha' would be marked VOC. Since news articles rarely address the reader directly, the Penn treebank contains very few instances of this tag (mostly in quotations).

## A.2 Semantic tags

This category includes those tags introduced in Sections 2.2.1 and 2.2.3 of the bracketing guidelines. They tend to mark the semantic role of a constituent, but there are few universal statements that can be made about their usage. In many cases they act adverbially, but even aside from the NOM tag they sometimes modify noun phrases or other things. Most mark adjuncts, but some mark complements.

### A.2.1 ADV—Adverbial

Constituents are marked ADV if they are not adverbial phrases, but are acting adverbially, but aren't marked with some other form/function tag. For instance, in "Sasha felt slightly sick", "slightly" is an adverb comprising an adverbial phrase, and therefore would not be tagged ADV. But in "Sasha felt a little bit sick", "a little bit" is a noun phrase acting adverbially, so it is tagged ADV. On the third hand, in "Sasha grew a little bit", "a little bit" is still modifying the verb, but it has a more specific tag to use (EXT), so it doesn't get an ADV tag.

### A.2.2 BNF—Benefactive

This tag marks an indirect object, in either shifted or unshifted form (see A.1.1), that uses or would use the preposition 'for'. That is, in the sentences "Alex made Sasha a pie" and "Alex made a pie for Sasha", "Sasha" and "for Sasha" (respectively) would carry the BNF tag.

### A.2.3 DIR—Direction

Constituents—usually prepositional phrases—that answer the questions "from where?" and "to where?" are marked with the DIR tag. (Those that answer the question "through where?" are marked LOC instead.)

As with many other tags, this includes metaphorical location, especially financial, as in "The price rose to $37", wherein "to $37" would be marked DIR.

### A.2.4 EXT—Extent

This marks the (usually financial 'pseudo-spatial') extent of an activity, as in "The price rose $4", where "$4" would be marked EXT.

Unlike most of the other tags in this category, EXT is rarely attached to a prepositional phrase, more usually marking noun phrases.

### A.2.5 LOC—Locative

This very common tag is used to mark phrases that denote the place where something takes place, usually as a VP modifier but also sometimes found on an NP modifier.

The location may be metaphorical, as in "panic in the market" or "banking at that company". However, idiomatic location, as in "under pressure", is not marked LOC.

If the location is a source or a destination, DIR may be more appropriate.

### A.2.6 MNR—Manner

This tag is used for phrases that indicate the manner in which some action was performed, or the instrument with which it was performed.

Some annotators liberally mark nearly every otherwise-unmarked ADVP as MNR, while others only mark phrases that actually indicate manner (see Section 7.6.3).

### A.2.7 NOM—Nominative

When headless relative clauses and gerunds are used nominally, they are marked NOM. (For instance, in "Baking pies is fun", "baking pies" is NOM.) Other non-NPs are not so tagged.

### A.2.8 PRP—Purpose

This tag marks constituents that annotate the purpose of or reason for an action. In the sentence "Sasha ran for health reasons", "for health reasons" would be marked PRP.

### A.2.9 TMP—Temporal

This is the most common tag in this category, and the second most common overall; it marks temporal constituents, those which answer the questions "when?", "how often?", and "how long?". Temporal phrases can be either noun phrases or prepositional phrases, but in a choice between marking a PP or its NP object, annotators are advised to mark the PP.[1]

## A.3 Topicalisation (TPC)

This tag marks elements that precede the subject in a declarative sentence. Topicalised elements must meet other criteria, which are given in the bracketing guidelines.

The reason this tag was separated from the other syntactic tags is that it can co-occur with several of them. Since the categories are comprised of otherwise mutually exclusive tags, it seemed a good idea to remove this one.

## A.4 Miscellaneous

These tags don't fit into the other categories.

### A.4.1 CLF—'It'-Cleft

Marks cleft constructions. In the sentence "It was Sasha that read the book", the top-level S node would be marked CLF.

[1] Except if the PP is already marked DIR and the sense of the expression is financial.

### A.4.2 CLR—"Closely related"

The CLR tag was meant to mark a variety of phenomena, most of which could be loosely grouped as collocation and idiom. The largest subgroup of CLR markings are phrasal verbs like 'rely on'. It turns out that this tag was ill-defined and never meant for public consumption; see Section 2.1.1.

### A.4.3 HLN—Headline

Headlines (and datelines) are marked with HLN; such constituents always occur at the top level, distinct from the following sentence.

### A.4.4 TTL—Title

This tag marks the titles of books, paintings, shows, and so on. Titles may be of any constituent type, and may be altogether ungrammatical—the rare labels NAC, NX, and X (all of which indicate unusual and difficult-to-bracket grammatical constructions) each occur at least once in the Penn treebank with TTL.

# Appendix B

# tsed

## B.1 The command-line

`tgrep [options]` *`spattern file(s)...`*

Prints to standard output those sentences in *`file(s)`* that match *`spattern`*. *`spattern`* is a search pattern constructed as described in Section B.2. Note that for most nontrivial search patterns you will need to put single quotes around the search pattern so that your shell does not interpret it as multiple arguments. Note also that in some shells (e.g. `tcsh`), some characters (e.g. `!`) will be grabbed by the shell even when single-quoted; this can be worked around with the alternate operators discussed in Section B.2.3.

`tsed [options]` *`spattern rpattern file(s)...`*

Prints to standard output the entirety of *`file(s)`*, with every sentence that matches *`spattern`* modified according to *`rpattern`*. *`spattern`* is a search pattern as for `tgrep`, described in Section B.2. *`rpattern`* is a replacement pattern, detailed in Section B.3.

`tsed [options] -b[`*`patfile`*`]` *`file(s)...`*

Instead of providing a single search-and-replace pattern on the command line, you can provide any number of them in a file (*`patfile`*, if present) or on standard input. Each line of the batch file contains one spattern (Section B.2) and one rpattern (Section B.3), each enclosed in curly brackets.[1] `tsed` will iterate through the patterns on a per-sentence basis, reading the sentence only once and printing it out when done.

`wsjsed [options]` *`file(s)...`*

When making a moderate number of changes to a corpus, or when making changes you'll want to communicate to someone else, you should wrap `tsed` with some program that understands the file structure of that corpus. For the Penn treebank, we have written `wsjsed`; it is described in Section B.4. Scripts to `wsjsed` can be provided as filenames on the command line or directly to

[1]The format is slightly more forgiving than that: any amount of whitespace can precede, follow, or separate the bracketed patterns, and the whole thing may be followed by a '`#`'-delimited comment. It must all fit on one line, however.

standard input.

### B.1.1 General options

Note that (for now) all options to `tgrep` and `tsed` need to be passed separately—"`tsed -C -g ...`" rather than "`tsed -Cg ...`"—for (not very good) historical reasons. This will be fixed in a future release.

`-D`*dir* The directory where internal data files reside. These include e.g. `headInfo.txt`, which is used to implement the head-finding / operator.

`-I`*dir* If present, this argument will be prepended to each input file provided.

`-d`*N* If $N$ is greater than zero, provides arcane debugging output.

`-C` Causes matching of leaf nodes (i.e. words) to be case-insensitive.

`-h` Provides a list of command-line options.

### B.1.2 `tgrep` options

`-f` Prints file and sentence number information for every match.

`-s` Instead of printing each match by itself, prints the entire sentence containing (at least) one match.

### B.1.3 `tsed` options

`-g` Replace all occurrences of a pattern. In batch mode, this option applies on a per-pattern basis; that is, the first spattern is matched and all occurrences are modified according to the rpattern before considering the second pattern.

`-G` Perform replacement, then repeat search-and-replace. This differs from the `-g` option in that for `-g` all the matches are found before any of the replacements are performed, and it is therefore guaranteed to terminate. The `-G` option, on the other hand, will re-match the spattern after the rpattern is applied, until no change results. Especially when used together with the `-b` option, it is very possible to cause infinite loops! The `-G` and `-g` can be used together, although in *most* (not all) cases the `-g` will be redundant.[2]

`-i` Perform change in place rather than outputting to standard output. Most corpus modification (e.g. `wsjsed`) will use this option.

[2]The application of the rpattern on an early match could cause a later spattern match to fail; if the `-g` option is used, all the spattern matches will be found already, and the rpattern applications won't affect them. If both `-g` and `-G` are used, all the spattern matches will be found, and the rpatterns will be applied; and then the whole thing will be repeated to see if any new spattern matches were formed.

`-c` Only print those sentences where at least one change was made. If a pattern fires but the modification is null (e.g. removing a tag that isn't there), the sentence won't be printed.

### B.1.4 `wsjsed` options

`-w`*path* Tells `wsjsed` where to find the copy of the Penn treebank to be modified. Default is the relative path '`wsj/`'.

## B.2 Search patterns

The search pattern syntax is based heavily on that used in Richard Pina's `tgrep` that comes packaged with the Penn treebank. There are some major differences, however, which will be noted below.

The basic structure of a search pattern is as follows:

*base* [*op arg*]*

The *base* of the pattern is a label or word to be searched for; a pattern with nothing but a base label will print every single tree that is rooted at a node with that label. To have a base match anything, use the underscore ('_') wildcard.

After the base you can have any number of *restrictions*, each of which are composed of an *operator* and an *argument*. The operators must be separated from the preceding pattern and from the succeeding argument by whitespace; this is because virtually any punctuation can actually occur in a word, and thus whitespace is our only consistent delimiter. (Also parentheses. More on that in a minute.) Some operators take atomic things, like numbers and labels, as their arguments; others can take patterns. Unless they are comprised solely of a base pattern, however, they must be enclosed by parentheses or square brackets, as the next unparenthesised operator will be taken as beginning a new restriction. For instance,

```
VP < NP < PP
```

searches for a VP that has at least one NP child and at least one PP child; on the other hand

```
VP < (NP < PP)
```

searches for a VP that has at least one NP child *which itself* has at least one PP child. Patterns can be nested indefinitely, and some pretty complex ones can arise; examples of this can be seen in Section B.2.4. The difference between parentheses and square brackets is irrelevant for `tgrep`; the use of square brackets in `tsed` spatterns is discussed in Section B.3.1.

It should be noted that search patterns are considered from left to right; for every constituent, it is first compared against the base pattern, then against each restriction in turn. If at any point a comparison fails, the remainder are not considered. It is therefore prudent to put any compute-intensive restrictions at the end of the pattern.

### B.2.1 Search pattern operators

Here we detail the behaviour of each operator, and say what kind of argument it can take.

**Sentence number: `#`**

This operator accepts a numeric argument $n$, and automatically fails unless the constituent under consideration is in sentence $n$ of the current input file (numbered from 0). Thus

```
S # 4
```

would find all S nodes in sentence 4 (the 5th sentence) of any of the input files. This will normally only be run on individual files, and is probably only useful in error-correction `tsed` patterns.

**Function tag: `-`**

This operator accepts a tag $x$, and succeeds if $x$ is among the hyphen-separated tags modifying the constituent under consideration.

```
PP - TMP
```

finds all PP nodes tagged with the TMP function tag.

**Head: `/`**

This operator accepts a word $w$, and succeeds if $w$ is the head word of the constituent under consideration. NB: the algorithm used to determine head words is specific to the Penn treebank tagset, and will work unpredictably or not at all with other tagsets!

```
NP / stock
```

finds the noun phrase(s) headed by the word stock.

Note that all heads are calculated when trees are read in, so that this restriction can be checked in constant time. Unfortunately, this imposes a penalty on spatterns that don't require it, so it may change in future versions (so that heads are only calculated if they will be used, for instance).

**Ancestor: `>>`**

This operator accepts a pattern $p$, and succeeds if any node on the path from the current constituent to the root node, including the root node but not the current constituent, is matched by the pattern $p$.

```
PP >> S
```

finds any prepositional phrase (PP) anywhere below an S node.

**Parent:** `>`

This operator accepts a pattern $p$, and succeeds if the parent of the node under current consideration is matched by $p$.

```
_ > PRN
```

finds all children of parentheticals (PRN).

**Child:** `<`

This operator accepts a pattern $p$, and succeeds if *any* child of the node under current consideration is matched by $p$.

```
PP < DT
```

finds all prepositional phrases that contain determiners at their top level.

**Specific child:** `<N`

This operator accepts a pattern $p$, and succeeds if the $N$th child of the node under current consideration is matched by $p$. An $N$ of 1 means the leftmost child. If $N$ is negative, count proceeds from the right, so that an $N$ of $-1$ means the rightmost child. The operators `<,` and `<‘` are synonyms for `<1` and `<-1`, respectively.

```
S <‘ NP
```

finds an S-node whose *last* child is a noun phrase.

**Descendant:** `<<`

This operator accepts a pattern $p$, and succeds if a node matching $p$ can be found anywhere in the subtree rooted at the node under current consideration.

```
S << PP
```

finds an S-node that has *any* PP descendants.

**Left descendant:** `<<,`

This operator accepts a pattern $p$, and succeeds if any left descendant of the node under current consideration matches $p$. A left descendant is any descendant that can be reached only by going through the leftmost children of each node in turn.

```
S <<, Each
```

finds any S-node whose first word is ‘Each’.

**Right descendant: »‘**

This operator accepts a pattern *p*, and succeeds if any right descendant of the node under current consideration matches *p*. A right descendant is any descendant that can be reached only by going through the rightmost children of each node in turn.

```
S «‘ ?
```

finds any S-node that ends with a question mark.

**Sibling: $**

This operator accepts a pattern *p*, and succeeds if any sibling of the node under current consideration matches *p*. Note that the siblings can occur in either order.

```
NP $ PRN
```

finds a noun phrase that is sibling to a parenthetical.

**Right-adjacent sibling: $.**

This operator accepts a pattern *p*, and succeeds if the sibling following the node under current consideration matches *p*. This operator is sensitive to order.

```
NP $. ,
```

finds a noun phrase that is followed by (not preceded by!) a comma-like punctuation mark. (That is, anything whose part-of-speech tag is a comma.)

**Precedent sibling: $..**

This operator accepts a pattern *p*, and succeeds if any sibling that precedes the node under current consideration matches *p*. The precedent sibling need not be adjacent.

```
PP $.. VP
```

finds a prepositional phrase preceded by a sibling VP.

### B.2.2 Search pattern negation: !

Any operator can be negated by preceding it with an exclamation point. Under the hood, this is implemented as simply running the restriction as if it were not negated, then reversing the sense of the result.

```
IN !> PP
```

finds prepositions not directly under a prepositional phrase.

### B.2.3 Alternate operator characters

Many patterns will be entered on the command line, but shells often do not play well with operator characters (notably '!'). As a result, there are some substitutions that can be made:

| | |
|---|---|
| N | ! |
| S | $ |
| P | < |
| C | > |

Table B.1: Operation substitution characters

These can be made on a character-by-character basis; for instance, legitimate substitutes for `!«` include `N«`, `NPP`, and even `NP<`.

### B.2.4 Search pattern examples

Here we list a few different sample search patterns, with a prose description of each.

#### Mislabelled logical subjects

```
NP > (PP - LGS)
```

This pattern finds all noun phrases whose parents are prepositional phrases marked as logical subjects. Note that since the base pattern is the NP, it is that node that will be printed out (not the parent PP). To print the PP instead, we would write the spattern as

```
PP - LGS < NP
```

or

```
PP < NP - LGS
```

which are semantically equivalent. The first will be somewhat faster, however, as restrictions are processed in order: the tag restriction string-compares "LGS" against every tag on that PP. If there are no tags, it fails immediately; in any case it's unlikely to have more than one tag to compare against. On the other hand, the child restriction needs to iterate through all children (always at least one, generally two or more) and string-compare "NP" against all the labels. This cost is more pronounced for constituent types that tend to have many children, and it's much much more pronounced for the descendant restriction (`«`), which has to check the entire subtree.

#### 'from... ago' phrases

```
PP / from «' ago
```

This pattern finds all prepositional phrases headed by 'from' and ending with 'ago'. The head test is cheap (see Section B.2.1), and the right-descendant test is not terribly expensive—linear in the depth of the subtree, rather than in the total size of the subtree. This won't get as many results as

```
PP / from « ago
```

but if you only care about phrases that *end* with 'ago', the former will be substantially more efficient.

## B.3 Replacement patterns

Once the pattern is matched, we need to know what to replace it with. The different types of replacement patterns tend to change a minimum of things about the matched pattern—no single rpattern can change both the label and the function tag of a node, for instance. As a result, a number of fairly simple changes will require multiple spattern-rpattern pairs to be applied in sequence. At some point there may be a mechanism for applying multiple rpatterns to a single spattern, but as there is already a batch capability in `tsed`, this is a relatively low priority.

### B.3.1 Referring to the matched spattern

There are several places in an rpattern where you might need to refer to something in the corresponding spattern. This is done by using square brackets [] instead of parentheses () in the spattern. The subpattern is referred to by a backslash followed by a number; the number corresponds to the sequential number of the square-bracketed subpattern. Such subpatterns are counted from 1, and in order by their left (open) bracket. Thus in a pattern like

```
A < [B < [C]] < [D]
```

we have `\1` referring to the B node, `\2` to the C node, and `\3` to the D node. The zeroth match, `\0`, refers to the entire spattern match (here the A node).

### B.3.2 The rpattern types

#### Removing nodes: `()`

To remove a matched node from the tree it is in, the rpattern is simply an empty pair of parentheses.

#### Relabelling nodes: *`LABEL`*

To change the label of the matched node, the rpattern is just the label to replace it with. Using this type of rpattern will not change the children of a given node, or any of the tags associated with it.

**Retagging nodes: `- TAG`, `! TAG`**

To add a tag to a node, use a hyphen, a space (that's important), and the tag to be added. To remove a tag, replace the hyphen with an exclamation point. Because it's possible to have multiple tags on a node, actually *changing* the tag is done by removing the old one and adding the new one—merely adding the new tag will preserve the old one intact. If a tag is to be added but is already present, no error is generated, but the program considers that no change is made (which may generate a warning down the line). Likewise, if a tag is to be removed but is not present, `tsed` considers that no change is made.

**Renumbering nodes: `- n`, `! n`**

To add or change the coreference index of a node, use a hyphen, a space, and the new coreference index. To remove the index, replace the hyphen with an exclamation point. It is not possible to have multiple coref indices for a single node, so adding a number to an already-indexed node will change the number. If an index is to be added and that $n$ is already the index of the node, the program considers that no change is made. If an index is to be removed and no index is present, or the index is different from $n$, no change is made.

**New subtrees: `(LABEL ...)`**

If the matched node is to be replaced with all-new subtrees that were not present in the original tree, the usual treebank S-expression format is used: an open paren, the label of the node, all the children of the new node, and a close paren. The children can themselves be "new" subtrees like this one, or "reference" or "reuse" subtrees (see below).

**Referencing subtrees: `\n`**

To include a submatch in its entirety with no alteration, just refer to it as described in Section B.3.1.

**Reusing subtrees: `(\n ...)`**

If there is a submatch of the spattern that contains a node of which you wish to change the children, you can *reuse* that node using this type of rpattern. It is just like a "new" subtree, except that the label is replaced with a reference to the node in the original tree; this causes not only the label but also the coref index and all function tags to carry over to the new tree. The children, however, are replaced with those provided in the rpattern. Said children can themselves be "new", "reference", or "reuse" subtrees.

### B.3.3 Replacing subpatterns

It sometimes happens to be easier to write an spattern where the base pattern is fairly high in the tree, but the node we wish to replace is further down, and therefore only a submatch of the spattern.

To handle these cases, it is possible to specify *which* subpattern is being modified by the rpattern. To do so, simply precede the rpattern with a number (no backslash!) and a space. If this is not done, zero is assumed, which corresponds to the node matched by the entire spattern.

### B.3.4 Some notes on the inner workings

In many respects, `tsed` acts very much like a combination of the `tgrep` tree-search utility and the `sed` text-replace (and more) utility. However, there is one important difference: when it handles references, it is actually handling them *by reference*. This is almost always what you would want, but may cause confusion in some cases. One consequent is that portions of trees cannot be copied—references can be used at most once each in the rpattern. If they are used more, the resulting behaviour is undefined.

Another, most noticeable when replacing subpatterns (cf Section B.3.3), is that nodes are removed before being reinserted. Thus, with an spattern like

```
A < [B] < [C < [D]]
```

(which would match a tree like (A B (C D E))), the rpattern

```
2 (\2 \1 \3)
```

would yield an output tree (A (C B D)). Note that although the initial 2 indicates that only the second subpattern (here labelled C) will be replaced, the first subpattern (B) is first removed from its place in the tree before being added back in as a child of C. (Also note that, due to the way the query was constructed, the E node just went away. That's a problem, we're working on it.)

### B.3.5 Example rpatterns

**Retagging**

```
- LOC
```

This pattern keeps the structure of the input tree exactly the same, except that the matched node is tagged LOC. If the spattern had matched (PP ...) the result would be (PP-LOC ...).

**Relabelling**

```
RB
```

This pattern will change the label of the matched tree to RB. If the spattern matched (IN down), the output would be (RB down).

**Adding structure**

```
1 (S \1 \2)
```

This rpattern assumes there were (at least) two square-bracketted submatches in the affiliated spattern; it pulls them out of their respective places in the tree and puts them under a newly-created S node, which in turn is placed where the first of the submatches originally was. Thus if the spattern had been

```
A < [B] < [C]
```

which had matched (A (B ...) (C ...) ...), the output tree would be (A (S (B ...) (C ...)) ...).

## B.4 Modifying the Penn treebank

The purpose for which `tsed` was written, and for which we envision it being used most often, was to provide a concise format with which to transmit modifications to large corpora—in particular, the Penn treebank. By itself, `tsed` can perform the modifications, but we need something a little more high-level to make modification scripts, and to handle the task of working specifically with treebank files. That program is `wsjsed`.

A script for `wsjsed` has, aside from blank lines and comments (lines whose first non-whitespace character is '#'), some number of `wsjsed` commands. Each of these is a directive to make a change to one sentence, one section, or the entirety of the Penn treebank, using calls to `tsed`. A command to `wsjsed` consists of three parts—the last two are the spattern and the rpattern, as described above.

The first part of the `wsjsed` command indicates which file(s) are to be modified, which sentences within those files, and any parameters to be passed to `tsed`, such as `-C` for case insensitivity. The exact format is

```
<section><file>#<sentence>[-<options>]
```

If there are no command line options, that portion can be omitted. To apply a change to all sentences in a given section, use the form

```
<section>*[-<options>]
```

To apply a change to the entire treebank, use

```
*[-<options>]
```

Note that the section and file number should always be two digits, even when less than 10 (pad with an extra zero if necessary). The sentence number can be any length.

### B.4.1 Comments

Outside of a pattern, the hash character (‘#’) marks the start of a comment; everything else on that line will be ignored. In the usual case, this means that a line that starts with ‘#’ will be treated as blank, and ignored.

### B.4.2 One-line commands

A one-line command has, predictably, all three parts on one line, each enclosed by curly brackets. The parts may be separated by whitespace. For instance, the line

```
{2421#5-C} {NP / stock} {VP}
```

in a `wsjsed` script would modify sentence 5 in file 21 of section 24 by making the NP headed by the word ‘stock’ (or ‘Stock’, or ‘STOCK’) into a VP.

### B.4.3 Batch commands

If multiple modifications are to be made to the same file, it is more efficient to put them into the same `wsjsed` command, using the batch syntax. To invoke the batch syntax, you need to provide the `-b` option. If this option is given, there should be an open curly bracket (‘{’) on the line, and nothing else (aside from whitespace). Subsequent lines are treated as part of the batch, and will be given verbatim to `tsed` (see Section B.1); the batch is terminated by a line whose only non-whitespace character is a close curly bracket (}).[3]

```
{2415#19-b} {
  {IN < in}        {RP}
  {ADVP / in}      {! DIR}
  {ADVP / in}      {PRT}
}
```

will modify sentence 19 of file 15 of section 24 of the treebank, by retagging the word ‘in’ as a particle, removing the DIR tag from the phrase headed by it, and relabel that phrase as a particle phrase.

## B.5 Installation

This is the section that needs the most work, because right now it is *extremely* ad hoc. There are many things that are ugly, requiring command-line configuration, that I’d eventually like to wrap into a relatively automatic “make install”-type script. For now, though, this is what there is.

[3]Yet another thing to be fixed eventually: make this a little less fragile with regards to what needs to be on which line.

### B.5.1 Executables

If you have one of the systems that I claim to be 'supported', you don't need to download the source. There are three executables: `tgrep`, `tsed`, and `wsjsed`. For the moment, in order for them to work, you need to also grab the data directory containing files like "`headInfo.txt`". This, again, is something that will be fixed up in future releases, but for now you need to get this directory and put it somewhere. Whenever you run any of the executables, you will need to provide that directory's location with the `-D` option. Thus

```
wsjsed -D/home/myname/tseddata/ scriptfile
```

is what you would need to invoke, assuming that's where you put the data directory.

You may need to edit the first line of `wsjsed` if your system's perl executable is someplace other than `/usr/local/bin`.

The other (minor) issue is, of course, where to put the executables themselves. It shouldn't matter, although you'll probably want to make sure they are in your PATH somewhere.

### B.5.2 Source

Here's where things get really hairy. I haven't been able to test this configuration on too many systems, so if you have problems, email me and we'll together work out what needs to be done.

You will need an ANSI-compliant C++ compiler (GNU's `g++` works admirably), and you will need `flex++` and `bison++`. The former comes installed on many systems, but you'll probably need to download the latter from the web—the sunsite mirrors have it. Finally, you will need to have `gmake` installed, although the primary make program (i.e. the one called when you simply type `make`) might be something else.

First, download and open the tarball. It will untar into a directory named `tsed`. That directory has a number of subdirectories; `data` is the directory mentioned above, that you'll need to provide `tsed` and the others with its path. `src` contains the source, obviously.

With any luck, you shouldn't need to play with anything in the subdirectories. The makefiles are set up in a pretty modular way, so that *hopefully* you will only need to create or modify the system-specific one. First, check what your ARCH environment variable is set to.[4] Edit the Makefile whose extension matches your ARCH: i686 for linux, etc. Probably the only thing you'll need to do is modify the locations of the various programs (`g++` et al).

Then, type '`make all`'. If it succeeds, `tgrep` and `tsed` will now be in the directory `$ARCH/O` (i.e. `ppc/O`, etc). `wsjsed` is in the main directory. Copy these three executables into some directory in your PATH.

If any of this doesn't work, email me—`dpb@cs.brown.edu`—and hopefully we can figure it out.

[4] Check by typing '`echo $ARCH`' into a shell. If it's not defined, you'll probably want to set it. The ones we have predefined are 'sun4' (for Sun/Solaris), 'i686' (for Linux), and 'ppc' (for Mac OS X). Yes, we know that this should be an OS variable. No, we're not going to fix it right now.

# Appendix C

# Treebank corrections

These corrections are also available on the web at http://www.cs.brown.edu/∼dpb/tbfix/.

Note that the misparse fix of 2428#2 was wrapped to fit the page, and should properly be on a single line.

```
#Type A
{24*-bg} {
  {NP !- LGS > (PP - LGS)} {- LGS}
  {PP - LGS} {! LGS}
}

#Type B
{2400#2}{NP < (NNP < York)}{- LOC}
{2400#2}{IN < about}{RB}
{2400#4}{IN < down}{RB}
{2400#6}{NP < (NNP < Wednesday)}{- TMP}
{2400#6}{ADVP / Meanwhile}{- TMP}
{2400#7}{PP / at << MMS}{- LOC}
{2400#11}{PP / in << imports}{- LOC}
{2400#14}{PP / in << prices}{- LOC}
{2400#14}{PP / in << consumer}{- LOC}
{2400#15}{PP / in << CPI}{- LOC}
{2400#16}{PP / among}{- LOC}
{2400#16}{PP / in << producer}{- LOC}
{2400#18-b} {
  {NP < (RB / as $. (_ / much) $.. (IN / as))} {QP}
  {NP / %} {- EXT}
}
```

```
{2400#20}{PP / in << inflation}{- LOC}
{2400#22}{PP / in << permits}{- LOC}
{2400#22}{PP / in << starts}{- LOC}
{2401#4}{IN < down}{RB}
{2402#2}{JJ < long}{VB}
{2402#7}{PP / in << detail}{- MNR}
{2402#8}{PP / in << room}{- LOC}
{2402#9}{IN < about}{RB}
{2402#14}{NP / morning}{- TMP}
{2402#21}{PP / on << globe}{- LOC}
{2402#23}{ADVP / still}{- TMP}
{2402#32-b} {
  {VP / make < [NP / him] < [ADJP / palatable]} {1 (S \1 \2)}
  {NP / him} {- SBJ}
  {ADJP / palatable} {- PRD}
}
{2402#34}{IN < about}{RB}
{2402#48}{PP / in << diaper}{- LOC}
{2403#2}{PP / to << Axa}{- DTV}
{2403#4}{IN < about}{RB}
{2403#12}{PP / in << states}{- LOC}
{2404#0}{IN < down}{RB}
{2404#1}{ADVP < NP < RBR}{- TMP}
{2404#2}{IN < down}{RB}
{2404#7}{IN < up}{RB}
{2404#12}{IN < down}{RB}
{2404#16}{PP / in << September}{- TMP}
{2404#27}{NP << between}{- EXT}
{2406#8}{SBAR <<, when << broadcasting}{- TMP}
{2406#32}{PP / Through}{- TMP}
{2406#36}{IN < around}{RB}
{2406#42}{NP / Newsreel}{- TTL}
{2407#5}{SBAR <<, because}{- PRP}
{2407#9}{PP / in << 1970s}{- TMP}
{2407#26}{NN < House}{NNP}
{2407#31}{ADVP / ever}{- TMP}
{2407#47}{PP / at}{- LOC}
{2409#0-g}{IN < about}{RB}
{2410#0}{PP / to << %}{- DIR}
```

```
{2410#2}{NP / quarter}{- TMP}
{2410#3}{IN < down}{RB}
{2411#3}{IN < about}{RB}
{2412#3-b} {
  {NP < [_ < Sen.] < [_ < Sam] < [_ < Nunn] < PRN} {1 (NP \1 \2 \3)}
  {NP / Ga} {- LOC}
  {NP / Okla.} {- LOC}
}
{2412#9}{PP / to << society}{- DTV}
{2412#33}{PP / about}{! LOC}
{2412#35}{PP / in << abstract}{! LOC}
{2412#50}{PP / in << U.S.}{- LOC}
{2412#69}{NP / behaviors}{- LGS}
{2412#69}{IN < down}{RP}
{2412#79}{SBAR <<, what}{- NOM}
{2413#9}{S / necessary}{FRAG}
{2413#21}{ADVP / ago}{- TMP}
{2413#23}{NP <<, Sunday}{- TMP}
{2413#27-b} {
  {IN < around} {RP}
  {ADVP / around} {! DIR}
  {ADVP / around} {PRT}
}
{2413#29}{S / to}{- PRP}
{2413#41}{ADVP / then}{- TMP}
{2415#0-b} {
  {IN < down} {RP}
  {ADVP / down} {! PUT}
  {ADVP / down} {PRT}
}
{2415#3}{PP / after}{- TMP}
{2415#4}{SBAR <<, what}{- NOM}
{2415#10}{NP / recovery}{- LGS}
{2415#17}{IN < about}{RB}
{2415#19-b} {
  {IN < in} {RP}
  {ADVP / in} {! DIR}
  {ADVP / in} {PRT}
}
```

```
{2415#45}{PP / in << account}{- LOC}
{2415#46}{IN < around}{RB}
{2416#7-b} {
  {PP / in << ownership} {- LOC}
  {NP / ownership} {! LOC}
}
{2417#4}{PP / in << volume}{- LOC}
{2417#15-b} {
  {VP / help < [NP / officials] < [VP / resolve]} {1 (S \1 \2)}
  {NP / officials} {- SBJ}
}
{2417#17}{SBAR <<, what}{- NOM}
{2417#20}{PP / After}{- TMP}
{2417#20}{IN < about}{RB}
{2417#22-b} {
  {IN < around} {RP}
  {ADVP / around} {! CLR}
  {ADVP / around} {! LOC}
  {ADVP / around} {PRT}
}
{2417#33-b} {
  {VP / having < [NP / trades] < [VP / flow]} {1 (S \1 \2)}
  {NP / trades} {- SBJ}
}
{2417#41}{NP / points}{- EXT}
{2417#45}{IN < about}{RB}
{2417#51}{PP / in << futures}{- LOC}
{2417#56}{NP / proposals < PRN}{- HLN}
{2417#66}{IN < about}{RB}
{2417#70}{IN < about}{RB}
{2417#71}{PP / at}{- LOC}
#{2417#73}{VBD < bribed}{VBN}
{2417#75}{IN < about}{RB}
{2417#84}{IN < about}{RB}
{2417#87}{IN < down}{RB}
{2417#87-b} {
  {SBAR < [S < (NP < [DT < that])]} {2 IN}
  {SBAR < [S < (NP < [IN < that])]} {(SBAR \2 \1)}
}
```

```
{2417#88}{DT < half}{NN}
{2417#88-b} {
  {NP > (PP / in << half)} {! TMP}
  {PP / in << half} {- TMP}
}
{2417#91}{IN < about}{RB}
{2418#12-g}{IN < about}{RB}
{2418#25}{ADJP / necessary}{- PRD}
{2418#25}{IN < on}{RB}
{2418#30}{ADVP / initially}{- TMP}
{2418#30}{IN < about}{RB}
{2418#34}{PP / In << papers}{- LOC}
{2418#37}{ADVP / so}{- PRD}
{2418#45}{PP / with << suit}{- MNR}
{2418#49}{PP / in << case}{- LOC}
{2419#2}{NP <<, Exeter << N.H.}{- LOC}
{2419#2}{NP <<, Fitchburg << Mass.}{- LOC}
{2422#0}{NP <<, New << York}{- LOC}
{2422#2}{NP <<, Tampa << Fla.}{- LOC}
{2422#2}{PP / for << year}{- TMP}
{2422#3}{S / is <<` basis}{- TPC}
{2424#3}{ADVP / ever}{- TMP}
{2424#3}{IN < about}{RB}
{2425#0}{IN < about}{RB}
{2426#6}{SBAR <<, which}{- NOM}
{2427#2}{PP / after << voting}{- TMP}
{2428#2}{VP < [VB < mark] < (S < [_ / down] < [NP / quotations])
    < [SBAR <<, while]}{(VP \1 \2 \3 \4)}
{2428#2-b} {
  {IN < down} {RP}
  {ADJP / down} {! PRD}
  {ADJP / down} {PRT}
}
{2428#4-b} {
  {S} {! TPC}
  {S} {! 1}
  {S < [NP / calamity] < [VP / is] < .} {1 (S \1 \2)}
  {S <<` over} {- TPC}
  {S <<` over} {- 1}
```

```
}
{2428#8}{NP < (CD < 2002)}{- TMP}
{2428#21}{IN < about}{RB}
{2428#23}{IN < about}{RB}
{2428#23-b} {
  {PP / to << 7.16} {! EXT}
  {PP / to << 7.16} {- DIR}
}
{2428#25}{SBAR <<, When}{- TMP}
{2428#46-b} {
  {(NP - ADV < (_ < iota))} {- EXT}
  {(NP - ADV < (_ < iota))} {! ADV}
}
{2428#49}{NP <<, Olympia}{- LGS}
{2428#51}{PP / before << Campeau}{- TMP}
{2428#52}{IN < down}{RB}
{2428#53}{NP / months}{- TMP}
{2428#54}{IN < about}{RB}
{2428#62-b} {
  {NP < [SBAR]} {\1}
  {SBAR <<, What} {- NOM}
  {SBAR <<, What} {- SBJ}
}
{2428#64-b} {
  {NP < [SBAR]} {\1}
  {SBAR <<, What} {- NOM}
  {SBAR <<, What} {- SBJ}
}
{2428#67}{NP / Thursday}{- TMP}
{2428#68}{PP / since << March}{- TMP}
{2428#75}{NP <<, about <<' %}{! EXT}
{2428#75}{IN < about}{RB}
{2428#76}{IN < about}{RB}
{2428#76}{IN < up}{RB}
{2428#81-b} {
  {NP < (NP < [RB < as] < [RB < much]) < (PP < [IN < as] < (NP < [CD] < [NN]))}
   {(\0 (QP \1 \2 \3 \4) \5)}
  {NP < QP} {- EXT}
}
```

```
{2429#0-b} {
  {VP / keep < [NP] < [ADJP]} {1 (S \1 \2)}
  {NP <<, markets} {- SBJ}
}
{2429#1}{IN < about}{RB}
{2429#2}{ADVP / immediately}{! TMP}
{2429#4}{ADVP / shortly}{! TMP}
{2429#23}{PP / under <<‘ pressure}{! LOC}
{2429#24}{PP / in <<‘ 1987}{- TMP}
{2431#0}{NP <<, this / year}{- TMP}
{2431#17}{ADVP / regularly}{- TMP}
{2431#26}{NP << coalition / birth}{- LGS}
{2431#29}{PP / in << shuffle}{- LOC}
{2431#29-b} {
  {PP / for << Greece} {! PRD}
  {ADJP < (JJ < Crucial)} {- PRD}
  {ADJP < (ADJP / Crucial)} {- ADV}
  {VP < [_ < are] < [PP / for]} {(\0 \1 (ADJP (-NONE- *?*)) \2)}
  {ADJP < -NONE-} {- PRD}
}
{2432#0}{PP <<, following}{- TMP}
{2432#6-b} {
  {IN < around} {RP}
  {ADVP / around} {! DIR}
  {ADVP / around} {PRT}
}
{2432#16}{IN < down}{RB}
{2432#16}{VBD < dragged}{VBN}
{2433#17}{PP / to << defunct}{- DTV}
{2433#23}{IN < about}{RB}
{2433#24}{IN < about}{RB}
{2433#36}{NP < [_ < about] < [_ < eight] < (_ < months)}{1 (QP \1 \2)}
{2433#36}{IN < about}{RB}
{2435#7}{IN < around}{RB}
{2436#5}{NP / today}{- TMP}
{2437#2}{IN < about}{RB}
{2438#0-g}{IN < about}{RB}
{2438#1-g}{IN < about}{RB}
{2438#3}{IN < about}{RB}
```

```
{2438#4}{PP / in << face}{! TMP}
{2438#6-g}{IN < about}{RB}
{2438#7-g}{IN < about}{RB}
{2438#8}{IN < about}{RB}
{2438#10-g}{IN < about}{RB}
{2438#11}{IN < about}{RB}
{2438#14-g}{IN < about}{RB}
{2440#3-g}{IN < about}{RB}
{2442#4}{NP <<, a / share}{- ADV}
{2442#4}{IN < about}{RB}
{2443#7}{SBAR <<, where}{- NOM}
{2443#28}{IN < about}{RB}
{2443#43-g}{IN}{RB}
{2443#50}{IN < down}{RB}
{2443#55}{PP / on << loans}{! LOC}
{2443#59}{PP / among}{- LOC}
{2444#11}{IN < down}{RB}
{2444#14-b} {
  {SBAR <<, Once} {! ADV}
  {SBAR <<, Once} {- TMP}
}
{2444#15}{NP << staggering / %}{- EXT}
{2444#20}{PP / in << years}{- TMP}
{2444#21}{IN < down}{RB}
{2444#21-b} {
  {PP / from << peak} {! TMP}
  {PP / from << peak} {- DIR}
}
{2444#37}{PP / in <<‘ 1983}{- TMP}
{2444#38}{NP / year <<‘ available}{- TMP}
{2444#41}{IN < down}{RB}
{2444#41-b} {
  {PP / from <<‘ peaks} {! TMP}
  {PP / from <<‘ peaks} {- DIR}
}
{2444#45}{PP <<, even / in}{- LOC}
{2444#45}{VP < (RB < even)}{ADVP}
{2444#48}{ADVP / ahead}{- TMP}
{2446#6}{PP / with <<‘ money}{- MNR}
```

```
{2446#15}{NP < [_ < about] < [_ < two] < (_ < weeks)}{1 (QP \1 \2)}
{2446#15}{IN < about}{RB}
{2446#16}{S / defrauding <<‘ Lincoln}{- NOM}
{2446#34}{SBAR <<, what << investigated}{- NOM}
{2448#1}{IN < about}{RB}
{2448#3}{PP / in << wake}{- LOC}
{2448#16-b} {
  {SBAR / while <<‘ billion} {! TMP}
  {SBAR / while <<‘ billion} {- ADV}
}
{2448#30}{IN < about}{RB}
{2448#35}{IN < down}{RB}
{2448#40}{PP / in <<‘ Guangdong}{- LOC}
{2449#3-b} {
  {PP / in << brief} {- LOC}
  {NP / brief  < VP} {! LOC}
}
{2450#0}{VBD < hit}{VBN}
{2450#3}{IN < around}{RB}
{2450#9}{ADVP / usually}{- TMP}
{2450#10}{IN < up}{RB}
{2450#12}{IN < down}{RB}
{2451#13}{SBAR <<, how}{- NOM}
{2451#13-b} {
  {IN < down} {RP}
  {ADVP / down} {PRT}
  {PRT - DIR} {! DIR}
}
{2451#15}{S > SINV}{- TPC}
{2451#15}{SBAR <<, what}{- NOM}
{2451#19}{PP / on << side}{- LOC}
{2451#26-b} {
  {JJ < unshackled} {VBN}
  {ADJP - PRD < [VBN] $ [S]} {(VP \1 \2)}
  {VP - PRD} {! PRD}
}
{2451#29}{NP / enterprises}{- LGS}
{2451#39}{PP / in <<‘ Washington}{- LOC}
{2453#1}{IN < down}{RB}
```

```
{2453#4}{IN < down}{RB}
{2453#6}{IN < down}{RB}
{2453#11}{IN < down}{RB}
{2454#9}{PP / in <<‘ Zambia}{- LOC}
{2454#11}{NN < back}{RB}
{2454#14}{NP / night}{- TMP}
{2454#20-b} {
  {SBAR <<, As} {! TMP}
  {SBAR <<, As} {- ADV}
}
{2454#24}{PP / under <<‘ pressure}{! LOC}
{2454#25}{S < (VP <<, to << see)}{- PRP}
{2454#25-b} {
  {SBAR <<, if} {! ADV}
  {SBAR <<, if} {- NOM}
}
{2454#29-b} {
  {VP / see < [S]} {1 NP}
  {NP - SBJ / lions} {! SBJ}
  {PP - PRD / in <<‘ action} {! PRD}
}
{2454#32}{PP / on <<‘ followers}{! LOC}
{2454#34}{ADVP / out}{PP}
```

# Appendix D

# The Principle of Indifference

The Principle of Indifference (or Insufficient Reason) is often stated as something like:

> Absent evidence to the contrary, all outcomes of a trial should be assumed to have equal probability.

It is virtually always attributed to Simon Pierre, Marquis de Laplace, though rarely with an actual cite; a few give his *Essai philosophique sur les probabilités* (1814) as the source. While this document remains a classic in the field (and an excellent read), the principle is not entirely original to this work, not named in this work, and not even really stated as a principle in itself, though it is noted in the discussion.

The first to enunciate the principle in some form was probably Jacques[1] Bernoulli, in his *Ars conjectandi* (1713) (emphasis mine):

> Similarly, the number of possible cases is known in drawing a white or a black ball from an urn, and one can assert that any ball is equally likely to be drawn; for it is known how many balls of each kind are in the jar, *and there is no reason why this or that ball should be drawn more readily than any other.*[2]

The connection between this citation and the Principle has been made in a number of places, but while the quote is definitely getting at the idea, it is made only in terms of the specific case under consideration. His nephew Daniel Bernoulli stated the principle in a more general form, and more clearly recognisable as a statement of the Principle in *Specimen theoriae novae de mensura sortis* (1738):

> Since there is no reason to assume that of two persons encountering identical risks, either

[1] a.k.a. James, Jakob, or Jacobi

[2] English translation found in (Calinger, 1995), created from the German translation by R. Haussner. In the original Latin: "Sic itidem noti sunt numeri casuum ad educendam ex urna schedulam albam nigramve, & notum est omnes æquè possibiles esse; quia nimirum determinati notique sunt numeri schedarum utriusque generis, *nullaque perspicitur traio, cur hæc vel illa potius exire debeat quàm quælibet alia.*"

> should expect to have his desires more closely fulfilled, the risks anticipated by each must be deemed equal in value.[3]

By comparison, Laplace (1814) leaves it mostly implicit in his First Principle of the Calculus of Probabilities: "The very definition of probability... is the ratio of the number of favourable cases to that of all possible cases."[4] Earlier he gives most of it in the midst of some discussion (emphasis mine):

> The theory of chances consists of reducing all events of the same kind to a certain number of equally possible cases, that is, cases *about whose existence we are equally uncertain*; and of determining the number of cases favourable to the event whose probability is sought. The ratio of this number to that of all possible cases is the measure of this probability, which is thus only a fraction whose numerator is the number of favourable cases, and whose denominator is the number of all possible cases.[5]

This is somewhat more clearly related to the Principle as we know it. Unquestionably, Laplace understood and believed in the underlying fact of the principle (as early as 1776: "if we see no reason why one case should happen more than the other" (Hacking, 1975, p. 131)). However, he never really states it as a principle in its own right, acknowledging that the important notion is not just equiprobability, but the indifference leading to the (presumed) equiprobability.

The first real statement of the Principle as such seems to come from Johannes von Kries. In his *Die Principien der Wahrscheinlichkeits-Rechnung* (1886), he states in Chapter I §4 (emphasis his):

> When now the logical [consequence] of our knowledge should present itself in the performance of a number of equally possible cases, thus arises without difficulty the explanation, *that two or more cases are to be regarded as equally possible, when in their respective circumstances we can find no reason to maintain one as possibly more probable than some other.*[6]

In the paragraph after this clear statement of the Principle, he names it (again, emphasis his):

> We want to briefly designate... that principle, on which the calculation of probability is based, as the *Principle of Insufficient Reason.*[7]

[3]Translation by (Sommer, 1954). Original not available.

[4]Translations from this work are based on (Dale, 1995). The original: "La définition même de la probabilité...est le rapport du nombre des cas favorables, à celui de tous les cas possibles."

[5]In the original French: "La théorie des hasards consiste à reduire tous les évènemens du même genre, à un certain nombre de cas également possibles, c'est-à-dire, tels que nous soyons également indécis sur leur existence; et à déterminer le nombre de cas favorable à l'évènement dont on cherche la probabilité. Le rapport de ce nombre à celui de tous les cas possibles, est la mesure de cette probabilité qui n'est ainsi qu'une fraction dont le numérateur est le nombre des cas favorables, et dont le dénominateur est le nombre de tous les cas possibles."

[6]All translations from this work are my own. In the original German: "Wenn nun das logische Verhalten unseres Wissens in der Aufführung einer Anzahl von gleich möglichen Fällen sich darstellen soll, so ergiebt sich ohne Schwierigkeit die Erklärung, *dass als gleich möglich zwei oder mehrere Fälle anzusehen sind, wenn in dem jeweiligen Stande unserer Kenntnisse sich kein Grund findet, unter ihnen einen für wahrscheinlicher als irgend einen anderen zu halten.*"

[7]The original German: "Wir wollen... das Princip, auf welches sie die Wahrscheinlichkeits-Rechnung basirt, als *Princip des mangelnden Grundes* [bezeichnen]."

Obviously, von Kries was aware of Laplace's work, and knew that the underlying implications were not original; he states as much when relating the history of probability theory in Chapter X, at the end of §3:

> With that, we reach essentially the point of view on which Laplace stands. In his writings, we find the short explanation: "equally possible cases, that is, cases about whose existence we are equally uncertain", a view in accordance with the Principle of Insufficient Reason that we have mentioned.[8]

Finally, early in the twentieth century, John Maynard Keynes gives the principle its now-more-familiar name in his *Treatise on Probability* (1921):

> The Principle of Indifference asserts that if there is no *known* reason for predicating of our subject one rather than another of several alternatives, then relatively to such knowledge the assertions of each of these alternatives have an *equal* probability. Thus *equal* probabilities must be assigned to each of several arguments, if there is an absence of positive ground for assigning *unequal* ones.

[8]The original German: "Hiermit ist im Wesentlichen der Standpunkt erreicht, auf welchem auch Laplace steht. Bei diesem finden wir die kurze Erklärung: "cas également possibles, c'est à dire tels que nous soyons également indécis sur leur existence," eine Auffassung, welche mit dem von uns so genannten Princip des mangelnden Grundes zusammentrifft."

# Bibliography

Lalit R. Bahl, Peter F. Brown, Peter V. de Souza, and Robert L. Mercer. 1989. A tree-based statistical language model for natural language speech recognition. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 37(7):1001–1008.

Collin F. Baker, Charles J. Fillmore, and John B. Lowe. 1998. The Berkeley FrameNet project. In *Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and Seventeenth International Conference on Computational Linguistics*, pages 86–90, Montréal.

Adam L. Berger, Stephen A. Della Pietra, and Vincent J. Della Pietra. 1996. A maximum entropy approach to natural language processing. *Computational Linguistics*, 22(1):39–71.

Jacques Bernoulli 1654–1705. 1713. *Ars conjectandi*. Published posthumously; author's first name also cited as James, Jacobi, Jakob; English translation of Part IV Chapter 4 appears in (Calinger, 1995).

Daniel Bernoulli 1700–1782. 1738. Specimen theoriae novae de mensura sortis. *Commentarii Academiae Scientiarum Imperialis Petropolitanae*, 5:175–192. Translated to English in (Sommer, 1954).

Ann Bies, Mark Ferguson, Karen Katz, and Robert MacIntyre, 1995. *Bracketing Guidelines for Treebank II Style Penn Treebank Project*, January.

Don Blaheta and Eugene Charniak. 2000. Assigning function tags to parsed text. In *Proceedings of the 1st Annual Meeting of the North American Chapter of the Association for Computational Linguistics*, pages 234–240.

Don Blaheta. 2002. Handling noisy training and testing data. In *Proceedings of the 7th conference on Empirical Methods in Natural Language Processing*.

Alena Böhmová, Jan Hajič, Eva Hajičová, and Barbora Hladká. in press. The Prague dependency treebank: a three-level annotation scenario. In Anne Abeille, editor, *Treebanks: Building and Using Syntactically Annotated Corpora*. Kluwer.

Thorsten Brants, Wojciech Skut, and Brigitte Krenn. 1997. Tagging grammatical functions. In *Proceedings of the 2nd conference on Empirical Methods in Natural Language Processing*.

Ronald Calinger, editor. 1995. *Classics of Mathematics*. Prentice Hall, Englewood Cliffs, N.J.

John Carroll and Ted Briscoe. 2001. High precision extraction of grammatical relations. In *Proceedings of the 7th International Workshop on Parsing Technologies*, Beijing.

John Carroll, Ted Briscoe, and Antonio Sanfilippo. 1998. Parser evaluation: a survey and a new proposal. In *Proceedings of the 1st International Conference on Language Resources and Evaluation*, pages 447–454, Granada, Spain.

Eugene Charniak. 2000. A maximum-entropy-inspired parser. In *Proceedings of the 1st Annual Meeting of the North American Chapter of the Association for Computational Linguistics*.

Michael Collins. 1996. A new statistical parser based on bigram lexical dependencies. In *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics*, Santa Cruz.

Michael Collins. 1997. Three generative, lexicalised models for statistical parsing. In *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics*, pages 16–23, Madrid.

Andrew I. Dale. 1995. *Philosophical Essay on Probabilities*. Springer-Verlag, New York. Translated from (Laplace, 1814).

Gregg Farnsborough, editor. 1967. *Specimen theoriae novae de mensura sortis*. Translation into English (Sommer, 1954) and German of (Bernoulli 1700–1782, 1738).

Yoav Freund and Robert E. Schapire. 1999. Large margin classification using the perceptron algorithm. *Machine learning*, 37(3):277–296.

Daniel Gildea and Daniel Jurafsky. 2002. Automatic labeling of semantic roles. *Computational Linguistics*, 28(3):245–288, September.

Jane Grimshaw. 1990. *Argument Structure*. MIT Press, Cambridge, MA.

Silviu Guiaşu and Abe Shenitzer. 1985. The principle of maximum entropy. *The mathematical intelligencer*, 7(1):42–48.

Ian Hacking. 1975. *The Emergence of Probability*. Cambridge University Press.

Edwin T. Jaynes. 1957. Information theory and statistical mechanics. *Physical Review*, 106:620–630. Reprinted in (Rosenkrantz, 1983).

Fred Jelinek and Robert L. Mercer. 1980. Interpolated estimation of Markov source parameters from sparse data. In Edzard S. Gelsema and Laveen N. Kanal, editors, *Pattern Recognition in Practice I*, pages 381–397, 401–402, Amsterdam. North-Holland Publishing Company (Elsevier).

John Maynard Keynes. 1921. *A Treatise on Probability*. Macmillan & Co., Ltd., London.

Simon Pierre le comte Laplace. 1814. *Essai philosophique sur les probabilités*. Mme. Ve. Courcier, Paris. Fifth edition published in 1825; English translation in (Dale, 1995).

David M. Magerman. 1994. *Natural language parsing as statistical pattern recognition*. Ph.D. thesis, Stanford University, February.

David M. Magerman. 1995. Statistical decision-tree models for parsing. In *Proceedings of the 33rd annual meeting of the Association for Computational Linguistics*.

Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz, 1995. *Penn Treebank II*. Linguistic Data Consortium, Philadelphia. LDC95T7.

Judita Preiss. 2003. Using grammatical relations to compare parsers. In *Proceedings of the 10th Conference of the European Chapter of the Association for Computational Linguistics*, Budapest.

Adwait Ratnaparkhi. 1999. Learning to parse natural language with maximum-entropy models. *Machine Learning*, 34:151–178.

Frank Rosenblatt. 1958. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65:386–408. Reprinted in *Neurocomputing* (MIT Press, 1998).

R.D. Rosenkrantz, editor. 1983. *E.T. Jaynes: Papers on Probability, Statistics, and Statistical Physics*. D. Reidel (Kluwer), Dordrecht, The Netherlands.

Claude E. Shannon. 1948. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423 and 623–656.

Louise Sommer. 1954. Exposition of a new theory on the measurement of risk. *Econometrica*, 22. Reprinted in (Farnsborough, 1967).

Hans Uszkoreit et al. 1997. Nebenläufige grammatische verarbeitung (NEGRA). http://www.coli.uni-sb.de/sfb378/negra-corpus/negra-corpus.html.

Johannes von Kries. 1886. *Die Principien der Wahrscheinlichkeits-Rechnung*. J.C.B. Mohr, Freiburg.