Refinements in Syntactic Parsing

David Hall

August 10, 2015

Refinements in Syntactic Parsing

by

David Leo Wright Hall

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Dan Klein, Chair
Line Mikkelsen
John DeNero

Summer 2015

Refinements in Syntactic Parsing

Abstract

Refinements in Syntactic Parsing

by

David Leo Wright Hall

Doctor of Philosophy in Computer Science

University of California, Berkeley

Dan Klein, Chair

Syntactic parsing is one of the core tasks of natural language processing, with many applications in downstream NLP tasks, from machine translation and summarization to relation extraction and coreference resolution. Parsing performance on English texts, particularly well-edited newswire text, is generally regarded as quite good. However, state-of-the-art constituency parsers produce incorrect parses for more than half of sentences. Moreover, parsing performance on other genres or in other languages is still quite hit-or-miss, mostly miss.

Many approaches have been developed for building constituency parsers over the years, including lexicalization (Collins 1997; Charniak 2000), structural annotation (Johnson 1998a; Klein and Manning 2003), and latent variable annotation (Matsuzaki, Miyao, and Tsujii 2005; Petrov et al. 2006). Each of these kinds of models have different strengths and weaknesses.

In this thesis, we develop a model of constituency parsing that attempts to unify these earlier parsing models. This approach centers around the idea of *refinement* of a simple underlying grammar. We show how latent variable modeling, structural annotation, and lexicalization can all be expressed as grammar refinements.

These kinds of refinements capture different phenomena, but systems have generally chosen one kind or another. This is because grammars can grow exponentially as additional refinements are added. We present an approach where multiple refinements coexist, but in a factored manner that avoids this combinatorial explosion. Our method works with linguistically-motivated annotations, induced latent structure, lexicalization, or any mix of the three. By using the approximate inference algorithm expectation propagation (Minka 2001), we are able to use many refinements at the same time, without incurring substantial overhead in terms of parsing speed. With this system in hand, we examine the extent to which using many kinds of refinements can improve performance.

From there, we question whether grammar refinements are necessary at all. We present a parser that largely eschews refinements in favor of simple surface configurations. We show that it works quite well on ten different languages from several different language families.

Finally, we show how to take advantage of the refinement structure of latent variable grammars to double the speed of an already extremely fast parser for graphics processing units (GPUs). Our resulting GPU parser is around 40 times faster than the original CPU implementation, parsing over 400 sentences per second on a commodity consumer-grade GPU.

To DeAnna

Contents

List of Figures

List of Tables

Acknowledgments

The LaTeX template here at Berkeley begins (and ends) the acknowledgments section with "I want to thank my advisor for advising me." Presumably, I am meant to replace that with a more lengthy and detailed tribute to my advisor, Dan Klein. But in some sense, anything longer would necessarily fall short, as any enumeration (Dan's brilliance, insight, dedication to his students, . . .) would of necessity would leave something left out (e.g. his teaching ability, writing, or emotional support). I would need some sort of 9th Amendment-style clause disclaiming the completeness, but such a statement would go under-appreciated, as they often do. So, I won't even try: Thanks, Dan, for advising me.

I've had the privilege of working directly with many of the members of the group, and they've all been a pleasure to work with. In no particular order: Taylor Berg-Kirkpatrick for the marathon problem set sessions in the early years in which I learned that vectors do in fact make sounds when they collide with a matrix, and later on for his insights and near limitless enthusiasm for experiments; David Burkett for all the time spent on Overmind together and the friendship that that led to; Greg Durrett, for his near-native speaker ability in Internetese and for bailing me out on at least one deadline; and Jono Kummerfeld for his awesome analysis tools and selflessly trekking to get us lunch each week.

I spent less time working on joint projects with the other members of the group—Jacob Andreas, Mohit Bansal, John Blitzer, John DeNero, Dave Golland, Aria Haghighi, Percy Liang, Adam Pauls, Max Rabinovich—but to the extent I overlapped with them they all contributed to making my time in grad school much more fun and much more stimulating. I'm particularly pleased that I'm now working with Jacob and Adam at my new venture. (Hopefully I'll get some of the rest of you later on!)

Even though I did not overlap with Slav Petrov, his influence on my work should be evident to anyone who spends any time with this thesis. I think I cite his papers around fifty times here.

There was a bit of a race I had with myself to see if I would get a thesis out of my parsing work or my historical linguistics work first. For better or for worse, parsing won. Nevertheless, the people I worked with on those projects were all fantastic to work with: Alexandre Bouchard-Côté, Chundra Cathcart, Will Chang, Andrew Garrett, Tom Griffiths, and Alex Nisnevich. In particular, Will's combined grasp of statistics, signal processing, computer science, and linguistics is nothing short of astounding.

Before Berkeley, I had the pleasure of spending my time at Stanford working in Dan Jurafsky and Chris Manning's NLP group. Throughout my PhD, they have both been great sources of advice and, when needed, encouragement. Thanks again for getting me started!

Similarly, Jason Eisner has been a tremendous inspiration, and I sincerely appreciate all the time he spent talking with me about random ideas at conferences and through email.

I would be remiss if I didn't thank my family who's been so supportive. From indulging me in my compulsive reading of C++ trade books (among many, many other things) as a teenager to letting me sneak away to California for college, and then grad school. . . and then life, they have been with me every step of the way.

And, of course, DeAnna, for everything. Everyone who's known her however briefly has had their lives markedly enriched, whether it be from caramels and ice cream or flower crowns and hand-made dog coats or her tireless Excel spreadsheeting. I am so incredibly fortunate, it is beyond any attempt at explanation on this page.

I am sure that I have left many people out who very much deserve mention here. Thank you, and please forgive my oversight.

A grammarian once embarked in a boat. Turning to the boatman with a self-satisfied air he asked him:

'Have you ever studied grammar?'

'No,' replied the boatman.

'Then half your life has gone to waste,' the grammarian said.

The boatman thereupon felt very depressed, but he answered him nothing for the moment. Presently the wind tossed the boat into a whirlpool. The boatman shouted to the grammarian:

'Do you know how to swim?'

'No' the grammarian replied, 'my well-spoken, handsome fellow'.

'In that case, grammarian,' the boatman remarked, 'the whole of your life has gone to waste, for the boat is sinking in these whirlpools.'

You may be the greatest scholar in the world in your time, but consider, my friend, how the world passes away - and time!

– Jalal al-Din Rumi. *Tales from Masnavi.* Trans. by A.J. Arberry.

Chapter 1

Introduction

This is a thesis about the syntactic parsing of natural language, which is to say analyzing the structure of natural language utterances. Syntax is in some real sense the distinguishing characteristic of human language. Other animals have sounds that they can communicate with: as I write this, one of my chickens is making her “I laid an egg” song, and, a short while ago, they made their danger noise when a neighborhood cat got too close. However, while they have a surprisingly rich vocabulary, chickens, like most animals, do not have a meaningful syntax for their language. That is, they do not compose small units of meaning (words, pieces of bird song) into arbitrarily larger units—as best we can tell, anyway. This is precisely what we do with syntax.

Because syntax is so important to human language, any system that can truly understand language must incorporate a model of syntax, either explicitly or implicitly. And so, just as syntax is essential to language, parsing is usually regarded as one of the “core” tasks of natural language processing. Parsing has many applications in “downstream” NLP tasks, from machine translation and summarization to relation extraction and coreference resolution. Parsing performance on English, particularly well-edited newswire text, is generally regarded as quite good. However, state-of-the-art parsers produce incorrect parses for more than half of sentences. Moreover, parsing performance on other genres or in other languages is still quite hit-or-miss, mostly miss.

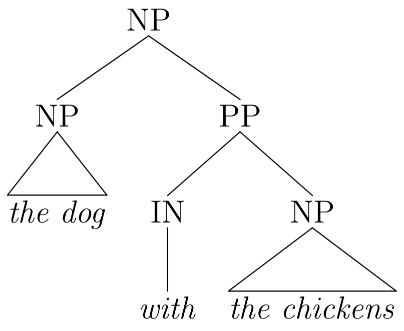
In this thesis, we develop an approach to constituency parsing that unifies many earlier parsing models based on refinements of a simple base grammar. We then define a new model that combines these different approaches in a way that is still computationally feasible. Further, we analyze the relationships between the different approaches to parsing, showing to what extent different refinements “stack” with one another when combined into a single system. Moreover, we present a radically simpler parsing system that achieves near or at state-of-the-art performance on 10 different languages, with nearly no adaptation. Finally, we will revisit refinements, this time in the context of making parsers faster. We will present a new system for parsing using Graphics Processing Units (GPUs) that exploits the structure imposed by refinements to make parsing faster.

1.1 Constituency

One of the key concepts in most theories of syntax is the *constituent*, a group of (consecutive) words in a sentence that functions as a single cohesive unit. Constituents include most of the categories one typically associates with grammar: noun phrases that make up the subject and objects of a sentence, prepositional phrases that modify other constituents, and embedded clauses that can either modify a constituent or be a complement.

Constituents are at the core of phrase structure grammars (Chomsky 1957), which are defined by "rules" that combine two or more constituents to form larger constituents. For example, the noun phrase "the dog with the chickens" is formed from a smaller noun phrase "the dog" and a modifying prepositional phrase "with the chickens". The phrase "with the chickens" can be further broken down into a preposition and its argument noun phrase. Schematically, this can be represented as a tree structure:

(1)



Many first cuts at syntax were based on context free grammars, or CFGs, which are one of the simplest ways for specifying phrase structure grammars. CFGs are composed of rewrite rules that expand a constituent symbol like "NP" into more constituents, or into "terminal" words. For example, the tree above would imply that we need a rewrite rule of the form NP → NP PP to represent the fact that an NP (like "the dog with the chickens") can be formed from combining an NP and a PP.

Consider the following sentence:

(2) The dog is eating the chickens.

A CFG that might produce this sentence is

(3) a. S → NP VP

b. NP → DT NN

c. VP → VB VP

d. VP → VB NP

e. DT → the

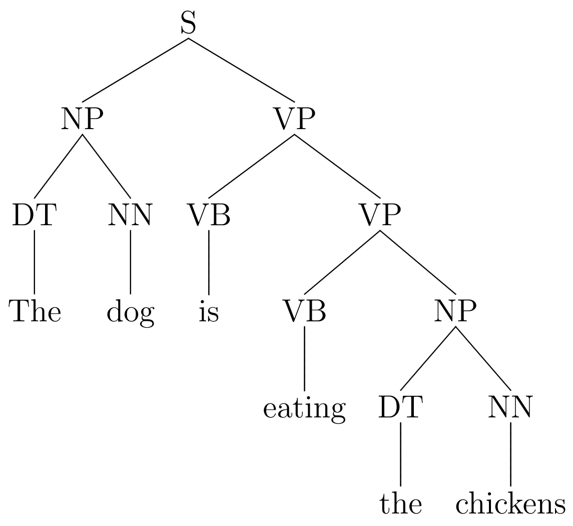
f. NN $\rightarrow$ dog

g. VB $\rightarrow$ is

h. VB $\rightarrow$ eating

i. NN $\rightarrow$ chickens

This CFG would yield this parse for (2):

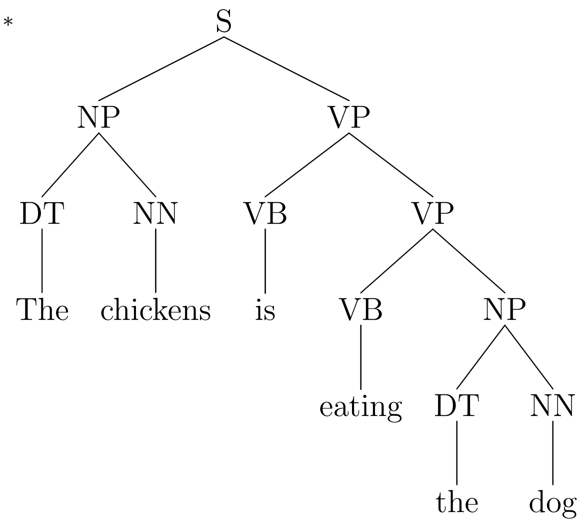
(4)


1.2 Refinements in Syntax

In practice, context free grammars do not work very well precisely because they lack context. For instance, consider this slight variation on (2):

(5) *The chickens is eating the dog.

The '*' indicates that this sentence is ungrammatical, while the first sentence is, of course, fine. However, in our simple grammar, these sentences are essentially identical in terms of their syntactic structure:

(6) *


The only difference—syntactically—between the two sentences is that "chickens" is plural while "dog" is singular. Therefore, we need to encode a notion of plural nouns into our grammar. The easiest way to do this is to "split" the NP symbol into different kinds, one for plural nouns and one for singular. That would give us symbols like NP[singular] and NP[plural]. We use the $X[r]$ notation to denote a *refinement* of the symbol X. Refinements provide a simple mechanism for enforcing relationships between the different symbols.

However, just annotating plurality is not enough. For example, we need to ensure that sentences like the following are also ungrammatical:

(7) *The dog eating the chickens.

This sentence does not have a finite verb (e.g. "is") So we need to augment our grammar to make the distinction between finite third person singular verbs (e.g. "is", "eats"), other finite forms of "to be" (e.g. "are"), and participles (e.g. "eating"):

(8) a. S → NP[3s] VP[3s]

b. S → NP[plural] VP[plural]

c. NP[3s] → DT NN[3s]

d. NP[plural] → DT NN[plural]

e. VP[3s] → VB[3s] VP[ing]

f. VP[plural] → VB[plural] VP[ing]

g. VP[plural] → VB[ing] VP

h. VP[3s] → VB[3s] NP

i. VP[plural] → VB[plural] NP

j. NP $\rightarrow$ NP[3s]

k. NP $\rightarrow$ NP[plural]

l. DT $\rightarrow$ the

m. NN[3s] $\rightarrow$ dog

n. VB[3s] $\rightarrow$ are

o. VB[plural] $\rightarrow$ are

p. VB[ing] $\rightarrow$ eating

q. NN[plural] $\rightarrow$ chickens

Historically, most statistical natural language constituency parsers have used refinements like these as their core representation, though sometimes they do so implicitly. Roughly speaking, all refinements can—like Gaul—be divided into three parts. Lexicalized parsers like those of Collins (1997) use a "head word" refinement that chooses a single word from each phrase. Structural parsers like those of Johnson (1998a) and Klein and Manning (2003) indirectly encodes information like "NP-modifying PP" versus "VP-modifying PP" by adding extra context to rules.[1] Finally, latent variable parsers like those of Matsuzaki, Miyao, and Tsujii (2005) and Petrov et al. (2006) use unsupervised learning to automatically allocate refinements in a way that best fits the data.

In the first part of this thesis, we will present a unified framework for representing these different kinds of refinements. Within this framework, we will describe three basic models representing each of these approaches. These models will serve as the basis for the more sophisticated models we consider later on.

1.3 Factored Refinements

While this refinements-based approach has been successful in past work in parsing, it has come at a computational cost. Each new refinement we add increases our grammar size multiplicatively. For example, distinguishing between plural and singular requires duplicating every noun phrase as well as every verb phrase. Further differentiating between nominative case pronouns like "I" and "he" versus accusative case "me" and "he" could double the number of noun phrases again. Languages with more complex morphology might also distinguish (and enforce agreement) between gender or more complex case relationships. As we consider more and more different phenomena, the size of our grammar will grow exponentially, which complicates both parsing and estimating the parameters for the parser.

[1]Collins (1997)'s parser also includes structural refinement in the form of verb subcategorization.

This explosion of rules and grammar symbols quickly becomes untenable, and so we will need another solution, for which we will turn to linguistic theory. Modern theories of syntax like Minimalism (Chomsky 1992) and Head-driven Phrase Structure Grammar(Pollard and Sag 1994) do not use the unanalyzed, monolithic categories like the ones we have defined so far. Rather they exploit the fact that these categories have a structure: there is a clear relationship between NP[3sing, +def] and NP[3sing, -def]. They analyze these categories into their component parts, which they call *features*. Features are then used to describe constraints on syntactic structures. For instance, we might have the constraints like "S $\rightarrow$ NP[+def] VP" and "S $\rightarrow$ NP[+3sing] VP[+present, +3sing]". The former requires that the subject of a sentence be definite, and the latter requires that the subject and verb agree if the verb is present tense and third person singular. Note that these are not discrete rules. Rather, they are *constraints* on any particular derivation, and they are applied in a factored way. (Minimalism calls this concept "checking;" HPSG uses a unification algorithm.)

Note that "feature" is a term of art in natural language processing and machine learning more broadly, where it refers to an individual predictor of the label of some datum. Thus, after this section we will refer to linguistic features as refinements, and we will use *factored refinements* when necessary to distinguish them from the monolithic categories.

In Chapter 5, we will spend some time discussing how linguistic theories use factored refinements. Then we will describe an approach to parsing that uses factored refinements in a way that is not dissimilar to the features used in linguistic theories. Moreover, we will show how to exploit the factored structure of these grammars to speed up parsing by introducing a parsing algorithm that is an instance of the approximate inference method expectation propagation (Minka 2001).

1.4 Parsing without Refinements

Refinements are a critical approach to most formal theories of syntax, and we have outlined some of the reasons why: distinguishing singular noun phrases from plural noun phrases is useful for telling ungrammatical sentences from grammatical ones, as is modeling the subcategorization frames of verbs.

On the other hand, from a practical perspective a parser does not need to distinguish between good sentences and bad sentences: we are given natural language text that is presumed to be good, and we need to recover the correct structure for it. Thus, *a priori* there is no reason that a parsing system should care about, for example, distinguishing singular from plural noun phrases.

Indeed, one could say this is the difference between natural language processing and computational linguistics (or perhaps merely discriminative versus generative modeling). In the former, we care more about getting the right answer than we do about getting the right answer for the right reason. In this thesis, we are engaged in the former, using the latter only as we find it useful. That is not to say that refinements are not useful—in fact, we will

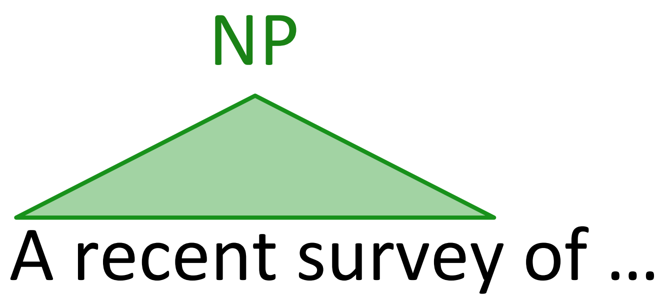


Figure 1.1: A simple noun phrase. Several context clues tell us the highlighted span "A recent survey" is a noun phrase, independent of our grammatical representation or formalism.

see that they are—but that we should carefully evaluate concepts from formal syntax for their impact on performance before including them in a practical system.

In the second part of the thesis, we will present a parser that largely eschews refinements in favor of simple surface configurations. Consider the example in Figure 1.1. How do we know that "A recent survey" is an NP?[2] For one, it begins with the word "A," which is usually a determiner, especially at the start of a sentence. The phrase also ends with a word that is quite likely to be a noun (though it could be a verb). It is also a relatively short phrase, which means it's probably not a full sentence. Finally, the word after the phrase is "of" which is a word that usually follows noun phrases. Taken together, the weight of these different features (in the machine learning sense) indicate that this span is probably a noun phrase. In Chapter 6, we will build a parser that exploits exactly this intuition with a very simple set of features, and we will show that it works quite well on nearly a dozen different languages, even languages that are known to have relatively free word order, like Hungarian.

The idea of using surface features itself is not novel. This approach is actually the predominant one in the dependency parsing literature (McDonald, Crammer, and Pereira 2005; McDonald and Nivre 2007; Koo, Carreras, and Collins 2008), and has seen some investigation within the constituency parsing literature. However, previous work in constituency parsing has also used surface features in their systems, but the focus has been on machine learning methods (Taskar et al. 2004), latent refinements (Petrov and Klein 2008a; Petrov and Klein 2008b), or implementation (Finkel, Kleeman, and Manning 2008). The contribution in this dissertation is our focus on a minimal grammar that uses no refinements, and the very simple set of features that we use.

In Section 6.5, we will consider combining refinements with surface configuration, showing experimentally which refinements "stack" with surface configurations.

[2]Linguists may be puzzled by my calling "A recent survey" a full NP, since there is a complementary PP that follows it. In the grammar of the Penn Treebank used, base constituents like this are treated as NPs.

1.5 Using Refinements for Parsing on GPUs

In the final chapter, we will shift our focus to a somewhat different topic: how can we make a really, really fast parser? We will describe a system for parsing using a Graphics Processing Unit, or GPU. GPUs are a substantially different architecture than modern CPUs: GPUs excel at performing the same operation in many places and accessing memory in a regular, structured way. Our refinement-free approach described in the previous section requires random accesses to memory, making it not a good candidate for GPUs.

Thus, the parser we will describe uses a grammar based on latent refinements (namely, the grammars from the Berkeley Parser (Petrov and Klein 2007)), and we will exploit this refinement structure to accelerate parsing performance by a factor of two relative to with a GPU approach that does not exploit that structure.

The end result is a parser can parse all of English Wikipedia (at time of writing) in just a few days on a single medium-end GPU, compared to the many CPU-months that a typical CPU implementation would take.

1.6 Contributions of This Dissertation

Our contributions are as follows. First, we build three models to serve as exemplars for the different kinds of refinements past authors have employed. Then, we introduce a new model for parsing with factored refinements, and an efficient approximate inference method for them using Expectation propagation. This model will allow us to build parsers with many different kinds of refinements simultaneously without creating an intractably large grammar. Next, we will examine the question of whether we need additional grammar refinements at all by building a very simple parser that uses only surface features and a basic grammar to achieve performance that is competitive with parsers that do rely on these carefully crafted refinements. We will then examine which refinements stack with this model. Finally, we will go back to refinements, showing how we can exploit the subcategorization structure to accelerate parsing on GPUs.

Chapter 2

Experimental Setup

The experiments in this thesis are mainly about parsing performance, either from an accuracy point of view (i.e. measuring the F1 score of a system's output on a particular corpus), or from a runtime point of view (measuring how long it takes to parse a corpus). In this chapter, we describe the experimental setup that we use for most of the experiments in this thesis.

In order to build (supervised) statistical models, we need a source of labeled data. In parsing, this comes in the form of a *treebank*, which is a corpus of sentences that have been annotated with syntactic parses.

2.1 English Language Experiments

For better or for worse, most NLP research is conducted on the English language, and much of this thesis will be no different; the majority of experiments we conduct will be on the English language.

The Wall Street Journal portion of the English Penn Treebank (Marcus, Santorini, and Marcinkiewicz 1993) has been the standard benchmark dataset for parsing for around twenty years, a testament to the size of its undertaking. It consists of about one million words of text from the Wall Street Journal annotated with constituency and part of speech tags. As it is the standard, we will use it as our main benchmark as we develop our models.

Unless otherwise noted, all experiments in this thesis will be conducted on the English Penn Treebank. We use sections 02-21 for training, 22 for the development set, and 23 for the test set. We report accuracies using EVALB (Sekine and Collins 1997) using a slight variation of the Collins parameter file, as is standard.

2.2 Multilingual Experiments

While most of our experiments will be on English language treebanks, it is also important to see how performance transfers to other languages. We will evaluate the performance of some of our systems on a collection of treebanks from 9 other languages: Arabic, Basque, French, German, Hebrew, Hungarian, Korean, Polish and Swedish. These treebanks were used in the 2013 Syntactic Parsing of Morphologically Rich Languages (SMPRL) shared task. (Seddah et al. 2013) Many (though not all) of these languages do in fact have rich morphologies. Some even have relatively free word order.

The data is derived from the following treebanks:

- **Arabic:** The Penn Arabic Treebank (Maamouri et al. 2003) and the Columbia Arabic Treebank (Habash and Roth 2009)
- **Basque:** The Basque Constituency Treebank (Aldezabal et al. 2008)
- **French:** The French Treebank (Abeillé, Clément, and Toussenel 2003)
- **German:** TiGer Treebank release 2.2 (Brants et al. 2002)
- **Hebrew:** The Modern Hebrew Treebank V2 (Sima'an et al. 2001)
- **Hungarian:** The Szeged Treebank (Csendes et al. 2005)
- **Korean:** The KAIST Treebank (Choi et al. 1994)
- **Polish:** The Składnica Treebank (Woliński, Głowińska, and Marek 2011)
- **Swedish:** The Talbanken section of the Swedish Treebank (Nivre, Nilsson, and Hall 2006)

The SPMRL treebanks come with automatically induced POS tags and morphological analyses. Preliminary experiments indicated that these were not useful for our system, and so we do not use them. They also provided a small training set condition, in which parsers are only given 5000 sentences of training data. We did not experiment with this condition.

Chapter 3

Parsing with Refinements

Many high-performance probabilistic constituency parsers take an initially simple base grammar over treebank labels like NP and enrich it with more complex syntactic features to improve accuracy. This broad characterization includes lexicalized parsers (Collins 1997), unlexicalized parsers (Klein and Manning 2003), and latent variable parsers (Matsuzaki, Miyao, and Tsujii 2005). Figures 3.1(a), 3.1(b), and 3.1(c) show small examples of context-free trees that have been annotated in these ways.

In this chapter, we will present a single canonical representation of constituency parsing that covers the grammars used by nearly all standard chart-based parsers—that is, leaving aside neural network parsers like Henderson (2003). In particular, we will frame these parsers as having *refinements* at their core. This abstraction will allow us to reason about the different ways these parsers work and to combine and manipulate those models to see how they interact.

3.1 Maximum Likelihood PCFGs

Before we think about building refined grammar, let us be more specific about exactly which grammar we are refining. The most obvious grammar one could build from a treebank is the maximum likelihood estimate (MLE) like that used in Johnson (1998b). In this grammar, we read off exactly the symbols used in the treebank, estimating the probability of each rule as proportional to the number of times we have seen it.

As an example, if we saw a configuration like the following, we would create a rule NP $\rightarrow$ DT NNP CD NN:

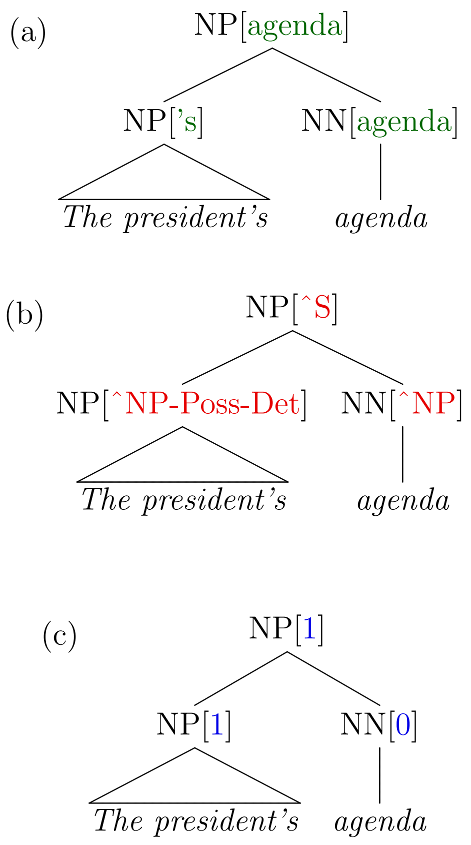
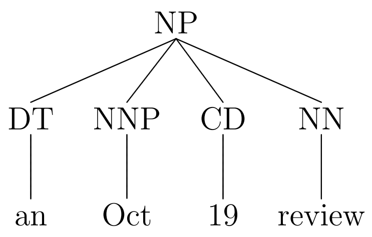


Figure 3.1: Parse trees using three different refinement schemes: (a) Lexicalized refinement like that in Collins (1997); (b) Structural refinement like that in Klein and Manning (2003); and (c) Latent refinement like that in Matsuzaki, Miyao, and Tsujii (2005).

(9)



Because we have to use a grammar in Chomsky Normal Form, we binarize the rules with more than two children by introducing synthetic symbols.[1] Our rule from above becomes

[1]There are many ways to binarize a rule. One could start by peeling off from the left hand side, or from the right hand side. We use "head outward binarization," where we pick a head symbol from among the children, and then add all symbols to its left, then to its right. For English, we use Collins (1997)'s head rules. See Section 3.5.

several:[2]

(10) a. NP $\rightarrow$ DT NP[\DT]

b. NP[\DT] $\rightarrow$ NNP NP[\NNP\DT]

c. NP[\NNP\DT] $\rightarrow$ CD NN

All in all, reading a grammar from the Penn Treebank training set produces a grammar with 38340 rules and 11681 symbols.[3]

Unfortunately, this grammar does not perform very well in practice, getting just 71.41 F1 on the development set of the Penn Treebank. The main reason this grammar does not work well is that the raw treebank symbols like NP do not encode enough information. For example, the maximum likelihood estimate grammar has no mechanism to differentiate the different rewrite statistics of subject NPs and object NPs (subject NPs are more likely to be pronomimal than object NPs), nor can it differentiate between PPs that attach to VPs (e.g. those headed by "to") and PPs that attach to NPs (e.g. those headed by "of"). These kinds of attachment problems are critical to parsing performance; having a grammar that can distinguish these "different kinds" of NPs and PPs is perhaps the most obvious way to deal with the problem.

However, the grammar actually encodes too much context in other ways. In particular, by using only configurations that have actually occurred in the treebank, the grammar cannot represent certain novel configurations. For example, no NP in the training set of the Penn Treebank has the rewrite "JJ NN VBG NNS," (e.g. "domestic printer manufacturing operations") but that configuration does occur in the development set.

One way to correct for this shortcoming is to *remove* even more information from the grammar in a process called "horizontal Markovization." Here, we shorten histories in the synthetic binarized symbols, removing a lot of the context that was encoded. For example, we might collapse the symbols NP[\NNP\DT] and NP[\NNP\JJ] into a single symbol NP[\NNP . . .], where the . . . mean that the symbol may have some number of predecessors, but the number and identity of those symbols are discarded. This has the effect of reducing the size of the grammar considerably, as well as allowing the grammar to produce more tree structures than it could before.[4] Removing all but 2 sibling gives a much smaller grammar with 3011 symbols and 16657 rules, scoring a slightly better 71.78 F1.

[2]The notation we use here is deliberately patterned on Categorial Grammar (Bar-Hillel 1953), where the categories of many words are functions that combine with other categories to form constituents. We use it merely to describe which neighbors a constituent has, but the binary rules produced by this process describe the order in which constituents combine, giving a similar interpretation, roughly speaking.

[3]Minor implementation differences will produce grammars with different numbers of rules.

[4]For the pedants: yes, the PCFG could already generate infinitely many trees. You know what I mean.

3.2 X-bar Grammars

But we can be more radical: we can strip the entire sibling histories from the binarized grammar collapsing all of the complex intermediate symbols like NP[\NNP\DT] into a symbol like "NP-bar." The resulting grammar would then have rules like NP $\rightarrow$ DT NP-bar. We call this grammar an "X-bar" grammar, because of its resemblance to the grammars from X-bar theory (Chomsky 1970; Jackendoff 1977) in linguistics.

This X-bar grammar is a much smaller grammar, with only 97 symbols (including synthetic symbols) and 4258 rules when extracted from the Penn Treebank. In a very real sense, this is the simplest grammar that makes any sense in the treebank: we include only those symbols that are in the unbinarized grammar, as well as the synthetic symbols (like "NP-bar") needed to reconstruct unbinarized trees from binarized trees.[5]

As one might imagine, this parser performs worse than the MLE grammar, scoring only 64.76 F1. Nevertheless, its simplicity makes it an excellent starting point for seeing how different kinds of refinements work; we will use it as the basis for all of our models.

3.3 Refinements

Equipped with our X-bar grammar, we can now proceed to describe refinements. Suppose we have a raw (binarized) X-bar grammar, with rules of the form A $\rightarrow$ B C. We will work with *refinements* of these rules, which are built from refined symbols that have been annotated with extra information, giving rules of the form A[x] $\rightarrow$ B[y] C[z]. An x might include information about the parent category, or a head word, or a combination of things. In the case of latent refinements, x will be an integer that may or may not correspond to some linguistic notion.

Take the process of horizontal Markovization. For the reasons we described before, horizontal Markovization has become a critical piece of nearly every parser. Typically, Markovization is described as a process of forgetting: we remove a certain amount (or all) of the history encoded in the binarized treebank symbols. However, if we start from an X-bar grammar, then horizontal Markovization of some of the past siblings can instead be interpreted as a addition to, or *refinement* of, the grammar, rather than as a subtraction. This is the framing we will follow in this thesis.

3.4 Basic Model

Now that we have a grammar, we need to specify a mathematical model. We employ a general exponential family representation of our grammar. This representation is fairly general, and—in its generic form—by no means new, save for the focus on refinements.

[5]One could imagine merging all synthetic symbols into a single "X-bar" category, or having the identity of the parent be independent of the identity of the children, or something else. We don't pursue that here.

Formally, we begin with a parse tree T over base symbols for some sentence $\mathbf{w}$, and we decorate the tree with refinements X, giving a parse tree $T[X]$. The conditional probability $\mathrm{P}(T[X]|\mathbf{w},\theta)$ of an annotated tree given words is:

$$\begin{aligned}
&\mathrm{P}(T[X]|\mathbf{w},\theta) \\
&\quad = \frac{f(T[X];\mathbf{w},\theta)}{\sum_{T',X'} f(T'[X'];\mathbf{w},\theta)} \\
&\quad = \frac{1}{Z(\mathbf{w},\theta)} f(T[X];\mathbf{w},\theta)
\end{aligned} \tag{3.1}$$

where the factor f takes the form:

$$f(T[X];\mathbf{w},\theta) = \exp\left(\theta^T \varphi(T,X,\mathbf{w})\right)$$

Here, φ is a feature function that projects the raw tree, refinements, and words into a feature vector. This model is thus a tree-valued conditional random field (Lafferty, McCallum, and Pereira 2001).

We further add a pruning filter that assigns zero weight to any tree with a constituent that a baseline unannotated X-bar grammar finds sufficiently unlikely, and a weight of one to any other tree. This filter is similar to that used in Petrov and Klein (2008a) and allows for much more efficient training and inference. Formally, it can be incorporated into the base measure of the distribution. We describe the pruning process we use in more detail in Section 3.5.

Because our model is discriminative, training takes the form of maximizing the probability of the training trees given the words. This objective is convex for deterministic refinements, but non-convex for latent refinements. We (locally) optimize the (non-convex) log conditional likelihood of the observed training data $(T^{(d)}, \mathbf{w}^{(d)})$:

$$\begin{aligned}
\ell(\theta) &= \sum_d \log \mathrm{P}(T^{(d)}|\mathbf{w}^{(d)},\theta) \\
&= \sum_d \log \sum_X \mathrm{P}(T^{(d)}[\mathrm{X}]|\mathbf{w}^{(d)},\theta)
\end{aligned} \tag{3.2}$$

Using standard results, the derivative takes the form:

$$\begin{aligned}
\nabla\ell(\theta) = &\sum_d E[\varphi(T,X,\mathbf{w})|T^{(d)},\mathbf{w}^{(d)}] \\
&- \sum_d E[\varphi(T,X,\mathbf{w})|\mathbf{w}^{(d)}]
\end{aligned} \tag{3.3}$$

The first half of this derivative can be obtained by the forward/backward-like computation defined by Matsuzaki, Miyao, and Tsujii (2005), while the second half requires an inside/outside computation (Petrov and Klein 2008a). The partition function $Z(\mathbf{w},\theta)$ is computed as a byproduct of the latter computation. Finally, we will regularize the objective.

For now, we note that we omit from our parser one major feature class found in other discriminative parsers, namely those that use features over the words in the span (Taskar et al. 2004; Finkel, Kleeman, and Manning 2008; Petrov and Klein 2008b). These features might condition on words on either side of the split point of a binary rule or take into account the length of the span. We will return to the question of features in Chapter 6.

3.5 Common Pre- and Post- processing

This section is full of details that are only interesting if you're trying to build your own parser.

We use a single preprocessing pipeline for all of our models and treebanks, with the exception that we convert the Arabic treebank—which is in the normalized form introduced by Buckwalter (2002)–into Unicode Arabic characters. We delete all traces and empty elements. In the Penn Treebank, there is one instance of the tag "PRT|ADVP," where evidently the annotators were not sure which one it was. We turn it into "PRT."

In the newer non-English Treebanks, it has become standard to annotate most punctuation as a single tag (e.g. "PU") rather than as the punctuation mark itself. We explicitly annotate tags over punctuation that do not match the punctuation with the punctuation itself.

Binarization

We use a variant of the standard CKY algorithm as our core dynamic program. One consequence of this choice is that we need to convert trees into Chomsky normal form (CNF), where each node in a tree has at most two children. We use a version Collins' head rules (Collins 1997) for our English language binarization.

For the foreign language experiments, we exploit the fact that all of the SPMRL treebanks have gold-standard dependency parses. Dependencies by their very nature describe head-dependent relationships, and so it is straightforward to convert a constituency tree and a corresponding (projective) dependency tree into a head-binarized constituency tree by identifying the head for each span as the word whose parent is outside of the span.[6]

Unary Rules

So far we have only discussed how we handle binary rules. However, treebanks typically have a number of unary productions, especially after traces and empty elements are stripped

[6]The SPMRL data we considered had no non-projective trees, but in that case, where there might be two or more words with parents outside of the span, we would arbitrarily choose one. Our algorithm currently chooses the one closest to the end of the constituent.

(as is customarily done). The correct handling of unary rules is a frequent challenge in the design of a parser. In principle, a derivation of a tree could use infinitely many unary rules, hopping from one constituent type to another.

One way to handle these infinite chains is to take the transitive closure of the graph induced by the unary rules. This operation computes the sum (or maximum) of the scores of all paths between all symbols using a version of the Floyd-Warshall algorithm (Roy 1959; Floyd 1962; Warshall 1962).

In practice, however, one does not see novel unary chains with much frequency (just two novel chains on the development set, out of more than 7500 unaries). If a particular path is never seen in the training data, it is unlikely to appear in the test data. Thus, allowing all possible transitions between any two constituents can only hurt performance.

We take a simple approach that captures this intuition. We only allow those paths in the unary graph that are actually seen in the training data. Rather than having unary rules for each link in a unary chain, we explicitly make each unique path its own unary rule. So, if we see the chain S $\to$ VP $\to$ VB, then we encode a single rule. But, if we have seen the unary rules S $\to$ VP and VP $\to$ NP but have never seen the chain S $\to$ VP $\to$ NP, we do not allow that as an output chain. As a consequence, every constituent in a tree has a "top" symbol and a "bottom" symbol representing the beginning and end of each unary chain. Because unary productions are not nearly as common as binary productions, most of the paths will be the length-0 path that represents an "identity" unary chain, and so these two symbols are usually identical. However, in the presence of a unary rule, these symbols are different, like S $\to$ VP. During parsing, the standard CKY parse chart is replaced with a two layer chart, one for the "bottom" symbol of a span, and another for the "top".

Decoding

Our model scores refined trees $T[X]$, while we are evaluated on unrefined trees. We could use the Viterbi one-best refined tree and simply drop the annotations. However, we can do better. Instead, we can pick a minimum Bayes' risk tree that maximizes some expected score of "goodness," marginalizing out over all possible refinements X for all possible trees.

There are several possible choices of MBR algorithms, each of which optimizes for a different criterion.[7] We use a fairly simple one: the Max-Recall algorithm of Goodman (1996). This algorithm maximizes the expected number of correct coarse symbols (A, i, j) with respect to the posterior distribution over parses for a sentence.

This particular MBR algorithm has the advantage that it is relatively straightforward to implement. In essence, we must compute the marginal probability of each refined symbol at each span $\mu(A[X], i, j)$, and then marginalize to obtain $\mu(A, i, j)$. Then, for each span (i, j), we find the best possible split point k that maximizes $C(i, j) = \mu(A, i, j) + \max_k (C(i, k) + C(k, j))$. Parse extraction is then just a matter of following back pointers from the root, as in the Viterbi algorithm.

[7]Many choices were explored in Petrov and Klein (2007).

Ordinarily, the Max Recall algorithm needs to be carefully tweaked to handle unaries; because it attempts to maximize the expected number of correct symbols (i.e. it does not penalize expected incorrect symbols), the algorithm should place every symbol in the grammar with non-zero marginal probability at every node in a long unary chain. In our model, there is exactly one unary rule at each cell, and so we avoid this problem.

We do, however, have to pick the best unary chain at each cell. Because unary chains are not explicitly represented in the parse chart, we have to infer them from the two end points (the "top" symbol A and "bottom" symbol B). To do so, we choose the chain the maximizes $O^T(A, i, j) f(\mathrm{A} \rightarrow \mathrm{B}_{\text{chain}}, i, j) I(B, i, j)$, where O^T and I^B are the outside top and inside bottom probabilities, respectively. Finally, when converting the tree into standardized form for output, we remove all identity unary chains.

Pruning

To make both training and parsing efficient, we use a simple coarse pruning approach in all of our experiments. We first construct a simple but inaccurate "coarse" grammar that can be parsed with quickly, and then we use it to filter out candidate constituents that have sufficiently low probability. Then, we use one of our more sophisticated "fine" models, skipping all candidate constituents (A, i, j) whose posterior probability under the simple grammar is less than e^{-7}. Coarse-to-fine approaches to parsing were pioneered by Charniak et al. (2006a) and thoroughly explored in the work of Petrov and Klein (2007). Our approach is similar to the latter's, but we use one only one coarse pass instead of several. (Using several would not make much sense for many of our models.)

Unless otherwise noted, we extract a simple generative X-bar grammar as our coarse grammar. This grammar is very weak—its error rate is around four times that of our best parsers—but nevertheless it can be used to filter out candidate constituents that are almost certainly not the correct answer.

Evaluation

In general, we use EVALB (Sekine and Collins 1997) to compute all parse evaluation metrics. When evaluating on the classical Penn Treebank, we follow standard practice and exclude punctuation from the evaluation. For all other datasets, we include punctuation, as is standard on those benchmarks.

Chapter 4

Three Parsing Models

In this chapter, we present three different kinds of parsing models that use different kinds of refinements. We first present a lexicalized model which annotates every constituent with a headword, a structural model that annotates constituents with its surrounding syntactic context, and a latent variable model that annotates each constituent with a latent cluster identifier. As we have said, there are alternative ways of building constituency parsers, for example, by either using surface features ((Taskar et al. 2004; Finkel, Kleeman, and Manning 2008; Petrov and Klein 2008b), or see Chapter 6), or neural networks (Henderson 2003; Socher et al. 2013a). Nevertheless, many of the most successful parsers use one of these models at their core.

As with most parsers or other complex natural language processing systems, the overall system is hard to completely describe in text, and so we are releasing our parser at `https://www.github.com/dlwh/epic`.

4.1 Structural Refinement

We have said that one of the reasons that raw X-bar grammar performs so badly is that it does not encode enough information to make correct decisions when parsing. For instance, the symbol PP for a prepositional phrases is used for both for those prepositional phrases that usually attach to noun phrases—such as those beginning with "of"—and for those that usually attach to verb phrases—such as those beginning with "to" or "by". This lack of modeling capacity is a direct consequence of the context-free grammar formalism: because decisions are local, a simple grammar cannot harness this critical contextual information.

One way to correct for this is to forcibly inject context into the grammar. That is, one could augment the raw symbols like PP with extra information about its environment: which constituents are above it in the tree, or to its left or right. For instance, we might annotate every PP with its parent constituent, so that now there are distinct symbols for prepositional phrases that have an NP parent ("PP[ˆNP]") and those that have a VP parent ("PP[ˆVP]").

This is the intuition exploited by parsers like the Johnson (1998b) and the Klein and

Manning (2003) parser. In their models, constituents are endowed with extra information describing their tree context, including parent information. Our structural parser is approximately a discriminative version of the generative model of Klein and Manning (2003).

This parser starts from a grammar with labels annotated with sibling, parent, and grandparent information, and then adds specific refinements, such as whether an NP is possessive or whether a symbol rewrites as a unary. More specifically, we use the following additional refinements:

1. **SplitAux**: Splits auxiliary verbs from other verbs. Specifically, inflections of "to be" are annotated with *AuxBe*, and all inflections of "to have" are annotated with *AuxHave*.

2. **SplitVP**: Infinitival VPs are split from finite verb phrases.

3. **SplitIN**: The preposition tag IN is split based on whether or not its grandparent is a noun phrase (NP), quantifier phrase (QP), embedded small clause (SBAR), or other embedded clause.

4. **SplitPossNP**: Possessive NPs are distinguished from non-possessives.

5. **MarkBaseNP**: Base NPs—those without phrasal modifiers or complements—are split from non-base NPs.

6. **MarkRightRecNP**: Marks NPs that are right-recursive: NPs whose right-most child is an NP.

7. **MarkNonIdentityUnaries**: Marks the parent of a unary rule if the parent and the child are not the same.

8. **MarkUnaryDTRB**: Marks determiners and adverbs that are the only word of a phrase.

9. **DominatesV**: Marks phrases that have some preterminal part of speech tag that is a verb.

Finally, like Klein and Manning (2003), we keep the temporal functional tag marker "TMP" that is used in the original Penn treebank to mark temporal noun phrases. All in all, this produces a grammar with 24024 symbols and 95667 rules. We experimented with slightly different refinements and adding more featurization, but this seemed to work about as well as the other slight variants we tried.

So far we have described the grammar for the structural model. We also have to describe what features we use. We use indicator features on the fully annotated rule, the rule with only parent and sibling refinement, the X-bar rule, and then features that conjoin the X-bar rule with the presence of each individual refinement (beside parent and sibling refinement) for each of the parent, left, and right children.

4.2 Latent Refinement

While structural refinement can be reasonably effective, many refinements require a lot of language-specific engineering. For instance, the **SplitIN** and **SplitAux** refinements are highly English-specific, requiring detailed knowledge of the kinds of constructions that occur in English. Moreover, even with language-specific knowledge, it is hard to know which refinements are likely to be important for good parsing performance.

An alternative to structural refinement that avoids these drawbacks is to use latent variables, wherein we ask a machine learning algorithm to divide constituents with the same symbol into clusters in a way that fits the data best. For instance, the algorithm might decide that NP-3 is for NPs that are modified by PPs, while NP-0 is for base NPs, and NP-1 is for NPs that usually do not take PPs (like pronouns). This approach was pioneered by Matsuzaki, Miyao, and Tsujii (2005), and significantly improved upon by Petrov et al. (2006).

Our latent variable model is essentially the same as the discriminative model of Petrov and Klein (2008a). We allocate some number of latent clusters to all constituents. Typically, in our model, we assign the same number of clusters to all constituents, with the exception that we assign only one cluster to symbols that the state-splitting Berkeley Parser (Petrov et al. 2006; Petrov and Klein 2007) assigns one cluster to when trained on the English Penn Treebank.[1] Those states are:[2]

- #
- $
- ”
- ,
- -LRB-
- -RRB-
- @CONJP
- @INTJ
- @LST
- @RRC
- @SBARQ
- @WHADJP
- @WHADVP
- @WHNP
- @X
- CONJP
- EX
- FRAG
- FW
- INTJ
- LS
- LST
- NAC
- PRT
- RBS
- ROOT
- RP
- RRC
- SBARQ
- SINV
- SYM
- TO
- UCP
- UH
- WDT
- WHADJP
- WHADVP
- WHPP
- WP
- WP$
- WRB
- X

Unlike the relatively complicated featurization of the structural model, we use only indicators on the fully annotated rule in our latent variable model. We found this to be more effective than including features on the unrefined X-bar rule as well.

[1]Of course, this is one particular execution of the Berkeley Parser, which has a randomized component. We used the publicly available grammar.

[2]Refer to the Penn Treebank II Standards Bies et al. (1995) for an explanation of these symbols. Currently they are summarized at `http://web.mit.edu/6.863/www/PennTreebankTags.html`. The “@” symbol denotes intermediate symbols produced by binarization, which we have called “-bar” previously.

4.3 Lexicalized Refinement

Recall that we said that we needed refinements to be able to distinguish between PPs that usually attach to noun phrases (like those that start with "of") and those that usually attach to verb phrases (like those that start with "to"). In the structural model, we captured that intuition by modeling tree context: we tell the grammar that PPs have different rewrite distributions depending on what their parent is. In the latent variable model, we asked the learning algorithm to learn which distinctions are most important.

In a lexicalized model, we instead model these attachment preferences directly using *head words*. A head word is the word that governs the syntactic behavior of a constituent. In a prepositional phrase, that word is the preposition. So, we can say there is a "PP[of]" and a "PP[to]." Because we are explicitly augmenting the grammar state with head words, many of the phenomena we would like to capture fall out directly. Lexicalized models were one of the earliest approaches to constituency parsing, and one of the most effective (Collins 1997; Charniak and Johnson 2005a).

As with our other models, our lexicalized model is a discriminative conditional random field parser. We are not aware of a prior discriminative lexicalized constituency parser, and it is quite different from generative models like Collins (1997). It is more similar to dependency parsers like McDonald, Crammer, and Pereira (2005); McDonald and Nivre (2007); Koo, Carreras, and Collins (2008). Broadly, our model considers features over a binary rule annotated with head words: A[h] $\rightarrow$ B[h] C[d] and A[h] $\rightarrow$ B[d] C[h]. We use the following features:

1. $\mathbb{I}[\text{A} \rightarrow \text{B C}]$
2. $\mathbb{I}[\text{A} \rightarrow \text{B C}, \text{HEAD} = \text{lfsuf}(w_h)]$
3. $\mathbb{I}[\text{A} \rightarrow \text{B C}, \text{DEP} = \text{lfsuf}(w_d)]$
4. $\mathbb{I}[\text{A} \rightarrow \text{B C}, \text{HEAD} = \text{lfsuf}(w_h), \text{DEP} = \text{lfsuf}(w_d)]$
5. $\mathbb{I}[\text{DIST}{=}\text{binDistance}(h, d), *]$
6. $\mathbb{I}[\text{A} \rightarrow \text{B C}, \text{HEAD-1} = \text{lfsuf}(w_{h-1})]$
7. $\mathbb{I}[\text{A} \rightarrow \text{B C}, \text{HEAD+1} = \text{lfsuf}(w_{h+1})]$
8. $\mathbb{I}[\text{A} \rightarrow \text{B C}, \text{DEP-1} = \text{lfsuf}(w_{d-1})]$
9. $\mathbb{I}[\text{A} \rightarrow \text{B C}, \text{DEP+1} = \text{lfsuf}(w_{d+1})]$

where tag maps each words to its most common tag, and lfsuf is the longest suffix of a word that occurs at least 100 times in the treebank. binDistance is a function that bins words into one of eight bins, based on their surface distance. The $*$ is meant to mean that we use all the previous features with their distance-augmented variants.

Model	F1
Latent, 2 states	81.7
Latent, 4 states	83.2
Structural Model	86.3
Lexicalized	87.3

Table 4.1: Baseline results for the three models. Experiments performed on the Penn Treebank development set (section 22). See Chapter 2 for a more detailed description of experimental parameters.

This is not quite a complete description of our features, as these templates would produce quite a large number of features. Instead, we produce these features only for configurations that actually occur in the treebank. For other "bad" features, we randomly hash their respective features to a number of buckets. We found that using a number of buckets equal to the number of good features worked well. (See Section 6.4 for a more detailed analysis.)

4.4 Experiments

We evaluated these baseline models on the development section of the Penn Treebank. We use Adaptive Gradient Descent (Duchi, Hazan, and Singer 2010) with L2 regularization for optimization, using a batch size of 512. We ran all experiments for 1000 gradient steps, which is about 10 passes through the Treebank. Performance usually leveled off after 3-4 passes; latent variables took longer.

Our strongest model is our lexicalized model, consistent with the usual results for English. Our structural model ties the generative model of Klein and Manning (2003) that it is based on. With two states (or one bit of annotation), our latent variable model gets 81.7 F1, edging out the comparable parser of Petrov and Klein (2008a). On the other hand, our parser gets 83.2 with four states (two bits), short of the performance of prior work.

Chapter 5

Factored Refinements

A preliminary version of this chapter appeared as Hall and Klein (2012).

In the previous chapter, we defined three different parsing models that used different kinds of refinements. While none of these models are quite state of the art for English, each of them performed reasonably well, with some models outperforming than others. In this chapter, we will present a model for working with all of these refinements at the same time in a way that is effective and computationally efficient.

Typically, parsers have focused on using one kind of refinement or another. Klein and Manning (2003) used only structural refinements, while Petrov et al. (2006) used only latent variables. As notable exceptions, the lexicalized model of Collins (1997) also included a fairly elaborate model of verb subcategorization, a form of structural refinement that is closely related to lexical refinement, while Charniak (2000) uses sibling, parent, and grandparent annotation in addition to lexicalization.

In general, when multi-part refinements are used in the same grammar, systems like that of Collins (1997) have generally multiplied these refinements together, in the sense that an NP that was definite, possessive, and VP-dominated would have a single unstructured PCFG symbol that encoded all three facts, like NP[ˆVP,+def,+POS]. In addition, modulo backoff or smoothing, that unstructured symbol would often have rewrite parameters entirely distinct from, say, the indefinite but otherwise similar variant of the symbol (Klein and Manning 2003). Therefore, when designing a grammar, one would have to carefully weigh new contextual refinements. Should a definiteness refinement be included, doubling the number of NPs in the grammar and perhaps overly fragmenting statistics? Or should it be excluded, thereby losing important distinctions? Klein and Manning (2003) discuss exactly such trade-offs and omit refinements that were helpful on their own because they were not worth the combinatorial or statistical cost when combined with other refinements.

Moreover, there has been to our knowledge no attempt to employ both latent and non-latent refinements at the same time. There is good reason for this: lexicalized or highly refined grammars like those of Collins (1997) or Klein and Manning (2003) have a very large number of states and an even larger number of rules. Further refining these rules with latent

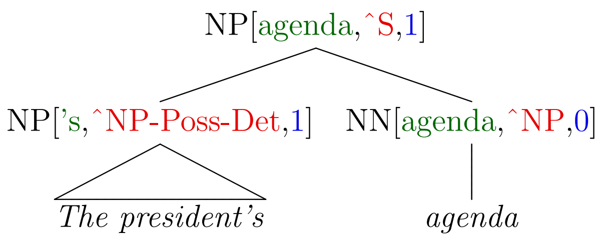


Figure 5.1: Parse trees using factored, mixed refinements. Compare with the examples of single kinds of refinements in Figure 3.1.

states would produce an infeasibly large grammar. Nevertheless, it is a shame to sacrifice expert refinement on the altar of unsupervised feature learning. Thus, it makes sense to combine these refinement methods in a way that does not lead to an explosion of the state space or a fragmentation of statistics.

In this chapter, we examine grammars with *factored refinements*, that is, grammars with refinements that have structured component parts that are partially decoupled. Our refined grammars can include both latent and explicit refinements, as illustrated in Figure 5.1, and we demonstrate that these factored grammars outperform parsers with unstructured refinements.

After discussing the factored representation, we describe a method for parsing with factored refinements, using an approximate inference technique called expectation propagation (Minka 2001). Our algorithm has runtime linear in the number of refinement factors in the grammar, improving on the naïve algorithm, which has runtime exponential in the number of refinements. Our method, the Expectation Propagation for Inferring Constituency (*Epic*) parser, jointly trains a model over factored refinements, where each factor naturally leverages information from other refinement factors and improves on their mistakes.

We demonstrate the empirical effectiveness of our approach in two ways. First, we efficiently train a latent-variable grammar with 8 disjoint one-bit latent refinement factors, with scores as high as 89.7 F1 on length $\leq$40 sentences from the Penn Treebank (Marcus, Santorini, and Marcinkiewicz 1993). This latent variable parser outscores the best of Petrov and Klein (2008a)'s comparable parsers while using two orders of magnitude fewer parameters. Second, we combine our latent variable factors with lexicalized and unlexicalized refinements, resulting in our best F1 score of 89.4 on all sentences.

5.1 Factored Refinements in Syntax

Linguists have long noted the limitations of CFGs as a grammatical formalism, essentially since their introduction. Chomsky (1965) noted that CFGs are unable to capture generalizations between what are plainly similar constituent types. As an example, Chomsky (1965) gives:

(11) a. N $\rightarrow$ Proper

b. N $\rightarrow$ Common

c. Proper $\rightarrow$ Proper-Human

d. Proper $\rightarrow$ Proper-non-Human

e. Common $\rightarrow$ Common-Human

f. Common $\rightarrow$ Common-non-Human

Under a CFG, there is no relationship between 'Common-Human' and 'Common-non-Human', which is clearly undesirable from a theoretical point of view. Chomsky (1965) thus proposed that CFGs be augmented with features. For instance, a noun phrase might have several features,[1] including one indicating whether or not the noun phrase refers to a human, and another indicating whether or not it is proper. Further, more syntactic features might be considered, like case or number.

In a similar vein, modern theories of grammar such as Minimalism (Chomsky 1992) and HPSG (Pollard and Sag 1994) do not ascribe unstructured conjunctions of refinements to phrasal categories. Rather, phrasal categories are associated with sequences of metadata that control their function. For instance, an NP might have refinements indicating that it is definite, singular, masculine, and nominative, with perhaps further information about its animacy or other aspects of the head noun.

Broadly speaking, these modern theories are centered around the amalgamation of two child constituents into a single larger parent. In unification-based formalisms, this operation is the eponymous unification operation.[2] In minimalism, this operation is called Merge.[3] We will use Amalgamate as a neutral term to emphasize our attempt to be theory-neutral.

Usually, this amalgamation satisfies some unsatisfied requirement of one of the children. This child is called the head. Schematically, we can represent it as follows:

(12)
$$\begin{tikzcd} & \mathrm{X}[\alpha \oplus \mathrm{Y}[\beta]] \arrow[dl, no head] \arrow[dr, no head] & \\ \mathrm{X}[\alpha, /\mathrm{Y}[\beta]] & & \mathrm{Y}[\beta] \end{tikzcd}$$

Here, X and Y are phrase structure categories like NP and VP, while α and β are bundles of features.[4]

[1] As we mentioned in the introduction of this dissertation, after this section we will refer to these "features" as "refinements" to avoid an ambiguity with the machine learning definition of "feature."

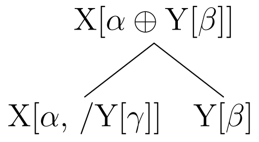
[2] Pollard and Sag (1994) actually use a slightly different operation called "conjunction."

[3] So capitalized. Some variants of Minimalism differentiate between Move and Merge.

[4] These theories actually go so far as to suggest that the categories like NP and VP are nothing more than features. It would be interesting to pursue that idea in a similar framework to ours, though we leave that for future work.

In Minimalism, the "/Y" is usually written as "uY," where the "u" means "uninterpretable"; a constituent with an uninterpretable feature cannot be present in a valid output structure. When the two constituents are merged, the uninterpretable feature is "checked" and then "deleted," indicating that is has been satisfied. The rest of the features of X pass through unchanged. That is, $\alpha \oplus \beta = \alpha$. Thus, the parent in (12) is just X[α].

In HPSG, features are not deleted. Instead, constraints are satisfied. As a consequence, this almagamation operation—induced by the Head Complement Rule or Head Specifier Rule—is somewhat more powerful. In particular, the "missing" part of the head child can be a partial specification of what is missing, rather than a single atomic feature. For instance, the missing part might just specify that the required constituent is a noun. Schematically, we have:

(13)


where $\beta \preccurlyeq \gamma$,[5] meaning that β is a structure that is a (potentially) more fully specified than γ, but that it has all of the features of γ, and no features that "disagree" with γ. As an example, γ might require that the Y is an animate noun phrase, but it might not care about whether or not the noun is plural or singular, masculine or feminine. The $\oplus$ operation here "unifies" the two bundles of features into a single larger bundle.

5.2 Model Representation

Thus, it is appealing for a grammar to be able to model the various, (somewhat) orthogonal factors that govern syntactic acceptability, that is, to have something like syntactic features. However, most models have no mechanism to encourage this. As a notable exception, Dreyer and Eisner (2006) tried to capture this kind of insight by allowing factored refinements to pass unchanged from parent label to child label, though they were not able to demonstrate substantial gains in accuracy.

In practice, our use of features (which we call "factored refinements") will be much more impoverished than any of the linguistic formalisms we described. In particular, we will not allow for complex, structured constraints like the kinds enabled by the $\preccurlyeq$ relationship we used for discussing HPSG. Instead, we will define a simple factored representation of a grammar into several components.

Suppose we have a raw (binarized) treebank grammar, with productions of the form A $\rightarrow$ B C. The typical process is to then annotate these rules with additional information, giving rules of the form A[x] $\rightarrow$ B[y] C[z]. In the case of explicit refinements, an x might include information about the parent category, or a head word, or a combination of things. In the case of latent refinements, x will be an integer that may or may not correspond to

[5]HPSG does not use this notation, but I find it useful.

some linguistic notion. We are interested in the specific case where each x is actually factored into M disjoint parts: $A[x_1, x_2, \ldots, x_M]$. (See Figure 5.1.) We call each component of x a *refinement factor* or a *refinement component*.

As in Chapter 3, we begin with a parse tree T over base symbols for some sentence $\mathbf{w}$, and we decorate the tree with refinements X, giving a parse tree $T[X]$. We focus on the case when X partitions into disjoint components $X = [X_1, X_2, \ldots, X_M]$. These components are decoupled in the sense that, conditioned on the coarse tree T, each column of the refinement is independent of every other column. However, they are crucially not independent conditioned only on the sentence $\mathbf{w}$. This model is represented schematically in Figure 5.2(a).

The conditional probability $\mathrm{P}(T[X]|\mathbf{w}, \theta)$ of an annotated tree given words is:

$$\begin{aligned}
&\mathrm{P}(T[X]|\mathbf{w}, \theta) \\
&\quad = \frac{\prod_m f_m(T[X_m]; \mathbf{w}, \theta_m)}{\sum_{T',X'} \prod_m f_m(T'[X'_m]; \mathbf{w}, \theta_m)} \\
&\quad = \frac{1}{Z(\mathbf{w}, \theta)} \prod_m f_m(T[X_m]; \mathbf{w}, \theta_m)
\end{aligned} \tag{5.1}$$

where the factors f_m for each model take the form:

$$f_m(T[X_m]; \mathbf{w}, \theta_m) = \exp\left(\theta_m^T \varphi_m(T, X_m, \mathbf{w})\right)$$

Here, X_m is the refinement associated with a particular model m. φ is a feature function that projects the raw tree, refinements, and words into a feature vector. The features φ need to decompose into features for each factor f_m; we do not allow features that take into account the refinements from two different components.

Because our model is discriminative, training takes the form of maximizing the probability of the training trees given the words. This objective is convex for deterministic refinements, but non-convex for latent refinements. We (locally) optimize the (non-convex) log conditional likelihood of the observed training data $(T^{(d)}, \mathbf{w}^{(d)})$:

$$\begin{aligned}
\ell(\theta) &= \sum_d \log \mathrm{P}(T^{(d)}|\mathbf{w}^{(d)}, \theta) \\
&= \sum_d \log \sum_X \mathrm{P}(T^{(d)}[X]|\mathbf{w}^{(d)}, \theta)
\end{aligned} \tag{5.2}$$

The derivative takes the form:

$$\begin{aligned}
\nabla \ell(\theta) &= \sum_d E[\varphi(T, X, \mathbf{w})|T^{(d)}, \mathbf{w}^{(d)}] \\
&\quad - \sum_d E[\varphi(T, X, \mathbf{w})|\mathbf{w}^{(d)}] \\
\nabla_{\theta_m} \ell(\theta) &= \sum_d E[\varphi_m(T, X_m, \mathbf{w})|T^{(d)}, \mathbf{w}^{(d)}] \\
&\quad - \sum_d E[\varphi_m(T, X_m, \mathbf{w})|\mathbf{w}^{(d)}]
\end{aligned} \tag{5.3}$$

The first half of this derivative can be obtained by the forward/backward-like computation defined by Matsuzaki, Miyao, and Tsujii (2005), while the second half requires an inside/outside computation (Petrov and Klein 2008a). The partition function $Z(\mathbf{w}, \theta)$ is computed as a byproduct of the latter computation.

5.3 Parsing with Factored Refinements

Note that the first term of Equation 5.3—which is conditioned on the coarse tree T—factors into M pieces, one for each of the refinement components. However, the second term does not factor because it is conditioned on just the words $\mathbf{w}$. Indeed, naïvely computing this term requires parsing with the fully articulated grammar, meaning that inference would be no more efficient than parsing with non-factored refinements.

Standard algorithms for parsing with a CFG run in $O(G|\mathbf{w}|^3)$, where $|\mathbf{w}|$ is the length of the sentence, and G is the size of the grammar, measured in the number of (binary) rules. Let G_0 be the number of binary rules in the unannotated "base" grammar. Suppose that we have M refinement components. Each refinement component can have up to A primitive refinements per rule. For instance, a latent variable grammar will have $A = 8^b$ where b is the number of bits of refinement. If we compile all refinement components into unstructured refinements (producing a grammar that looks something like Example 11), we can end up with a total grammar size of $O(A^M G_0)$, and so in general parsing time scales exponentially with the number of refinement components. Thus, if we use latent refinements and the hierarchical splitting approach of Petrov et al. (2006), then the grammar has size $O(8^S G_0)$, where S is the number of times the grammar was split in two. Therefore, the size of annotated grammars can reach intractable levels very quickly, particularly in the case of latent refinements, where all combinations of refinements are possible.

Dreyer and Eisner (2006) suggested an iterative approach to splitting refinements that were sufficiently different from each other. Similarly, Petrov et al. (2006) introduced a heuristic approach to merging grammar refinements that were not sufficiently different to provide any gains in likelihood. Thus, their approaches tried simply to slow down the exponential growth of splitting. In a follow-up, Petrov and Klein (2007) then presented a hierarchical coarse-to-fine inference algorithm that gave large increases in parsing speed.

Petrov (2010) considered an approach to slowing this growth down by using a set of M independently trained parsers P_m, and parsed using the product of the scores from each parser as the score for the tree. This approach worked largely because training was intractable: if the training algorithm could reach the global optimum, then this approach might have yielded no gain. However, because the optimization technique is local, the same algorithm produced multiple grammars.

In what follows, we propose another solution that exploits the factored structure of our grammar with expectation propagation. Crucially, we are able to jointly train and parse with all refinement factors, minimizing redundancy across the models. While not exact, we will see that expectation propagation is indeed effective.

5.4 Factored Inference

The key insight behind the approximate inference methods we consider here is that the full model is a product of complex factors that interact in complicated ways, and we will approximate it with a product of corresponding simple factors that interact in simple ways. Since each refinement factor is a reasonable model in both power and complexity on its own, we can consider them one at a time, replacing all others with their approximations, as shown in Figure 5.2(c).

The way we will build these approximations is with *expectation propagation* (Minka 2001). Expectation propagation (EP) is a general method for approximate inference that generalizes belief propagation. We describe it here, but we first try to provide an intuition for how it functions in our system. We also describe a simplified version of EP, called *assumed density filtering* (Boyen and Koller 1998), which is somewhat easier to understand and rhetorically convenient. For a more detailed introduction to EP in general, we direct the reader to either Minka (2001) or Wainwright and Jordan (2008a). Our treatment most resembles the former.

Factored Approximations

Our goal is to build an approximation that takes information from all components into account. To begin, we note that each of these components captures different phenomena: a structurally-refined grammar is good at capturing structural relationships in a parse tree (e.g. subject noun phrases have different distributions than object noun phrases), while a lexicalized grammar might capture preferred attachments for different verbs. At the same time, each of these component grammars can be thought of as a refinement of the raw unannotated treebank grammar. By itself, each of these grammars induces a different posterior distribution over *unannotated* trees for each sentence. If we can approximate each model's contribution by using only unannotated symbols, we can define an algorithm that avoids the exponential overhead of parsing with the full grammar, and instead works with each factor in turn.

To do so, we define a sentence specific *core approximation* over unannotated trees $q(T|\mathbf{w}) = \prod_m \tilde{f}_m(T, \mathbf{w})$. Figure 5.2(b) illustrates this approximation. Here, $q(T)$ is a product of M structurally identical factors, one for each of the annotated components. We will approximate each model f_m by its corresponding $\tilde{f}_m$. Thus, there is one color-coordinated approximate factor for each component of the model in Figure 5.2(a).

There are multiple choices for the structure of these factors, but we focus on *anchored PCFGs*. Anchored PCFGs have productions of the form $_{\mathrm{i}}\mathrm{A}_{\mathrm{j}} \to {_{\mathrm{i}}\mathrm{B}_{\mathrm{k}}}\, {_{\mathrm{k}}\mathrm{C}_{\mathrm{j}}}$, where i, k, and j are indexes into the sentence. Here, $_iA_j$ is a symbol representing building the base symbol A over the span $[i, j]$.

Billott and Lang (1989) introduced anchored CFGs as "shared forests," and Matsuzaki, Miyao, and Tsujii (2005) have previously used these grammars for finding an approximate one-best tree in a latent variable parser. Note that, even though an anchored grammar is unannotated, because it is sentence specific it can represent many complex properties of the full grammar's posterior distribution for a given sentence. For example, it might express a preference for whether a PP token attaches to a particular verb or to that verb's object noun phrase in a particular sentence.

Before continuing, note that a pointwise product of anchored grammars is still an anchored grammar. The complexity of parsing with a product of these grammars is therefore no more expensive than parsing with just one. Indeed, anchoring adds no inferential cost at all over parsing with an unannotated grammar: the anchored indices i, j, k have to be computed just to parse the sentence at all. This property is crucial to EP's efficiency in our setting.

Assumed Density Filtering

We now describe a simplified version of EP: parsing with assumed density filtering (Boyen and Koller 1998). We would like to train a sequence of M models, where each model is trained with knowledge of the posterior distribution induced by the previous models. Much as boosting algorithms (Freund and Schapire 1995) work by focusing learning on as-yet-unexplained data points, this approach will encourage each model to improve on earlier models, albeit in a different formal way.

At a high level, assumed density filtering (ADF) proceeds as follows. First, we have an initially uninformative q: it assigns the same probability to all unpruned trees for a given sentence. Then, we factor in one of the annotated grammars and parse with this new augmented grammar. This gives us a new posterior distribution for this sentence over trees annotated with just that refinement component. Then, we can marginalize out the refinements, giving us a new q that approximates the annotated grammar as closely as possible without using any refinements. Once we have incorporated the current model's component, we move on to the next annotated grammar, augmenting it with the new q, and repeating. In this way, information from all grammars is incorporated into a final posterior distribution over trees using only unannotated symbols. The algorithm is then as follows:

- Initialize $q(T)$ uniformly.
- For each m in sequence:
  1. Create the augmented distribution $q_m(\mathbf{T}[X_m]) \propto q(\mathbf{T}) \cdot f_m(\mathbf{T}[X_m])$ and compute inside and outside scores.
  2. Minimize $\mathrm{D}_{\mathrm{KL}}\left(q_m(T)||\tilde{f}_m(T)q(T)\right)$ by fitting an anchored grammar $\tilde{f}_m$.
  3. Set $q(T) = \prod_{m'=1}^{m} \tilde{f}_{m'}(T)$.

Step 1 of the inner loop forms an approximate posterior distribution using f_m, which is the parsing model associated with component m, and q, which is the anchored core approximation to the posterior induced by the first $m-1$ models. Then, the marginals are computed, and the new posterior distribution is projected to an anchored grammar, creating $\tilde{f}_m$. More intuitively, we create an anchored PCFG that makes the approximation "as close as possible" to the augmented grammar. (We describe this procedure more precisely in Section 5.4.) Thus, each term f_m is approximated in the context of the terms that come before it. This contextual approximation is essential: without it, ADF would approximate the terms independently, meaning that no information would be shared between the models. This method would be, in effect, a simple method for parser combination, not all that dissimilar to the method proposed by Petrov (2010). Finally, note that the same inside and outside scores computed in the loop can be used to compute the expected counts needed in Equation 5.3.

Now we consider the runtime complexity of this algorithm. If the maximum number of refinements per rule for any factor is A, ADF has complexity $O\left(MAG_0|\mathbf{w}|^3\right)$ when using M factors. In contrast, parsing with the fully annotated grammar would have complexity $O\left(A^M G_0|\mathbf{w}|^3\right)$. Critically, for a latent variable parser with M refinement bits, the exact algorithm takes time exponential in M, while this approximate algorithm takes time linear in M.

It is worth pausing to consider what this algorithm does during training. In training, we seek to maximize the probability of the correct tree T given the words $\mathbf{w}$. At each step, we have in q an approximation to what the posterior distribution looks like with the first $m-1$ models. In some places, q will assign high probabilities to spans in the gold tree, and in some places it will not be so accurate. θ_m will be particularly motivated to correct the latter, because they are less like the correct tree. On the other hand, θ_m will ignore the other "correct" segments, because q has already sufficiently captured them.

Expectation Propagation

While this sequential algorithm gives us a way to efficiently combine many kinds of refinements, it is not a fully joint algorithm: there is no backward propagation of information from later models to earlier models. Ideally, no model should be privileged over any other. To correct that, we use EP, which is essentially the iterative generalization of ADF.

Intuitively, EP cycles among the models, updating the approximation for that model in turn so that it closely resembles the predictions made by f_m in the context of all other approximations, as in Figure 5.2(c). Thus, each approximate term $\tilde{f}_m$ is created using information from all other $\tilde{f}_{m'}$, meaning that the different refinement factors can still "talk" to each other. The product of these approximations q will therefore come to act as an approximation to the true posterior: it takes into account joint information about all refinement components, all within one tractable anchored grammar.

With that intuition in mind, EP is defined as follows:

- Initialize contributions $\tilde{f}_m$ to the approximate posterior q.
- At each step, choose m.

    1. Include approximations to all factors other than m: $q^{\backslash m}(T) = \prod_{\mathrm{m}' \neq \mathrm{m}} \tilde{f}_{m'}(T)$.

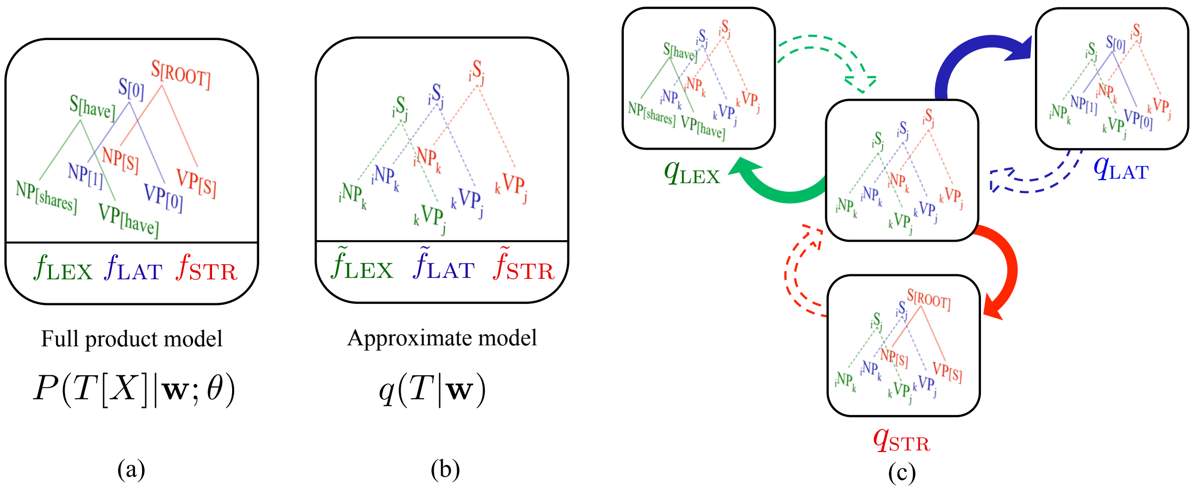


Figure 5.2: Schematic representation of our model, its approximation, and expectation propagation. (a) The full joint distribution consists of a product of three grammars with different refinements, here lexicalized, latent, and structural. This model is described in Section 3.2. (b) The core approximation is an anchored PCFG with one factor corresponding to each refinement component, described in Section 5.1. (c) Fitting the approximation with expectation propagation, as described in Section 5.3. At the center is the core approximation. During each step, an "augmented" distribution q_m is created by taking one refinement factor from the full grammar and the rest from the approximate grammar. For instance, in upper left hand corner the full f_{LEX} is substituted for $\tilde{f}_{\text{LEX}}$. This new augmented distribution is projected back to the core approximation. This process is repeated for each factor until convergence.

2. Create the augmented distribution by including the actual factor for component m
   $$q_m(T[X_m]) \propto f_m(T[X_m])q^{\setminus m}(T)$$
   and compute inside and outside scores.
3. Create a new $\tilde{f}_m(T)$ that minimizes $\mathrm{D}_{\text{KL}}\left(q_m(T)||\tilde{f}_m(T)q^{\setminus m}(T)\right)$.

- Finally, set $q(T) \propto \prod_m \tilde{f}_m(T)$.

Step 2 creates the augmented distribution q_m, which includes f_m along with the approximate factors for all models except the current model. Step 3 creates a new anchored $\tilde{f}_m$ that has the same marginal distribution as the true model f_m in the context of the other approximations, just as we did in ADF.

Assumed density filtering is a special case of EP in which only a single pass is made over the models. Moreover, the technique of Petrov (2010) can be seen as a degenerate variant of EP where the $\tilde{f}_m$ are set independently.

In practice, it is usually better to not recompute the product of all $\tilde{f}_m$ each time, but instead to maintain the full product $q(T) \propto \prod_m \tilde{f}_m$ and to remove the appropriate $\tilde{f}_m$ by division. This optimization is analogous to belief propagation, where messages are removed from beliefs by division, instead of recomputing beliefs on the fly by multiplying all messages.

Schematically, the whole process is illustrated in Figure 5.2(c). At each step, one piece of the core approximation is replaced with the corresponding component from the full model. This augmented model is then reapproximated by a new core approximation q after updating the corresponding $\tilde{f}_m$. This process repeats until convergence.

Epic Parsing

In our parser, EP is implemented as follows. q and each of the $\tilde{f}_m$ are anchored grammars that assign weights to unannotated rules. The product of anchored grammars with the annotated factor f_m need not be carried out explicitly. Instead, note that an anchored grammar is just a function $q(\mathrm{A} \to \mathrm{B\ C}, i, k, j) \in \mathbb{R}^+$ that returns a score for every anchored binary rule. This function can be easily integrated into the CKY algorithm for a single annotated grammar by simply multiplying in the value of q whenever computing the score of the respective production over some span. The modified inside recurrence takes the form:

$$\begin{aligned}
&\textsc{inside}(A[x], i, j) \\
&\quad = \sum_{B,y,C,z} \theta^T \varphi(\mathrm{A[x]} \to \mathrm{B[y]\ C[z]}, \mathbf{w}) \\
&\qquad\quad \cdot \sum_{i<k<j} \textsc{inside}(B[y], i, k) \cdot \textsc{inside}(C[z], k, j) \\
&\qquad\qquad\quad \cdot q(\mathrm{A} \to \mathrm{B\ C}, i, k, j)
\end{aligned} \tag{5.4}$$

Thus, parsing with a pointwise product of an anchored grammar and an annotated grammar has no increased combinatorial cost over parsing with just the annotated grammar.

To actually perform the projection in step 3 of EP, we create an anchored grammar from inside and outside probabilities.[6] Given a set of inside and outside scores over annotated labels $I(A[x], s, t)$ and $O(A[x], s, t)$, we compute the expected number of times the rule ${}_{\mathrm{i}}\mathrm{A}_{\mathrm{j}} \to {}_{\mathrm{i}}\mathrm{B}_{\mathrm{k}}\ {}_{\mathrm{k}}\mathrm{C}_{\mathrm{j}}$ occurs, and then then we locally normalize for each symbol ${}_iA_j$, using the equations in Figure 5.3. Basically, we just marginalize over refinements, and normalize so that $\sum_{B,C,u} p({}_sA_t \to {}_sB_{uu}C_t) = 1$.

This procedure actually creates the new q distribution, and so we have to divide out $q^{\backslash m}$. This process minimizes KL divergence subject to the local normalization constraints.

All in all, this gives an algorithm that takes time $O\left(IMAG_0|\mathbf{w}|^3\right)$, where I is the maximum number of iterations, M is the number of models, and A is the maximum number of refinements for any given rule.

[6]This discussion largely derives from Matsuzaki, Miyao, and Tsujii (2005)'s , where they used an anchored grammar for decoding.

$$\begin{aligned}
p({}_sA_t \to {}_sB_{uu}C_t) &= \frac{\sum_{xyz} O(A[x],s,t)\text{score}(\text{A[x]} \to \text{B[y] C[z]}, \mathbf{w}, s, t, u) I(B[y],s,u) I(C[z],u,t)}{\sum_x O(A[x],s,t) I(A[x],s,t)} \\
p({}_sA_t \to {}_sB_t) &= \frac{\sum_{xy} O(A[x],s,t)\text{score}_u(A \to B, \mathbf{w}, s, t) I(B[y],s,t)}{\sum_x O(A[x],s,t) I(A[x],s,t)} \\
p({}_sA_{s+1} \to w_s) &= 1
\end{aligned}$$

$$\begin{aligned}
\text{score}(\text{A[x]} \to \text{B[y] C[z]}, \mathbf{w}, s, t, u) &= \exp\left(\theta_m^T \varphi(\text{A[x]} \to \text{B[y] C[z]}, \mathbf{w}) + \log q(\text{A} \to \text{B C}, s, u, t)\right) \\
\text{score}_u(A[x] \to B[y], \mathbf{w}, s, t) &= \exp\left(\theta_m^T \varphi(A[x] \to B[y], \mathbf{w}) + \log q(A \to B, s, t)\right)
\end{aligned}$$

Figure 5.3: Estimating an anchored grammar for a sentence $\mathbf{w}$ from the inside and outside scores of an anchored grammar. Note that we use alternating layers of unaries and binaries.

Other Inference Algorithms

To our knowledge, expectation propagation has been used only once in the NLP community; Daumé III and Marcu (2006) employed an unstructured version in a Bayesian model of extractive summarization. Therefore, it is worth describing how EP differs from more familiar techniques.

EP can be thought of as a more flexible generalization of belief propagation, which has been used several times in NLP (Smith and Eisner 2008; Niehues and Vogel 2008; Cromières and Kurohashi 2009; Burkett and Klein 2012). In particular, EP allows for the arbitrary choice of messages (the $\tilde{f}_m$), meaning that we can use structured messages like anchored PCFGs.

Mean field (Saul and Jordan 1996) is another approximate inference technique that allows for structured approximations (Xing, Jordan, and Russell 2003; Burkett, Blitzer, and Klein 2010), but here the natural version of mean field for our model would still be intractable. However, it is possible to adapt mean field into allowing for tractable updates that are similar to the ones we proposed. We do not pursue that approach here.

Dual decomposition (Dantzig and Wolfe 1960; Komodakis, Paragios, and Tziritas 2007) has recently become popular in the community (Rush et al. 2010; Koo et al. 2010). In fact, EP can be seen as a particular kind of dual decomposition of the log normalization constant $\log Z(\mathbf{w}, \theta)$ that is optimized with message passing rather than (sub-)gradient descent or LP relaxations. Indeed, Minka (2001) argues that the EP objective is more efficiently optimized with message passing than with gradient updates. This assertion should be examined for the structured models common in NLP, but that is beyond the scope of this dissertation.

Finally, note that EP, like belief propagation but unlike mean field, is not guaranteed to converge, though in practice it usually seems to. In our experiments, typically three or four iterations are enough for almost all sentences to reach convergence, and we found no loss in cutting off the number of iterations to four.

	Parsing			
Training	ADF	EP	Exact	Petrov
ADF	84.3	84.5	84.5	82.5
EP	84.1	84.6	84.5	78.7
Exact	83.8	84.5	**84.9**	81.5
Indep.	82.3	82.1	82.2	82.6

Table 5.1: The effect of algorithm choice for training and parsing on a product of two 2-state parsers on F1. Petrov is the product parser of Petrov (2010), and Indep. refers to independently trained models. For comparison, a four-state parser achieves a score of 83.2.

5.5 Experiments

In what follows, we describe four experiments. First, in a small experiment, we examine how effective the different inference algorithms are for both training and testing. Second, we scale up our latent variable model into successively larger products. We also will look at how expectation propagation's runtime compares to exact inference for these models. Finally, we present a selection of the many possible model combinations, showing that combining latent and expert refinement can be quite effective.

Experimental Setup

For our experiments, we trained and tested on the Penn Treebank using the standard splits described in Section 2.1. Each discriminative parser was trained using the Adaptive Gradient variant of Stochastic Gradient Descent (Duchi, Hazan, and Singer 2010). Smaller models were seeded from larger models. That is, before training a grammar of 5 models with 1 latent bit each, we started with weights from a parser with 4 factored bits. Initial experiments suggested this step did not affect final performance, but greatly decreased total training time, especially for the latent variable parsers. When using EP or ADF, we initialized the core approximation q to the uniform distribution over unpruned trees.

When counting parameters, we consider the number of parameters per binary rule. Hence, a single four-state latent model would have 64 ($= 4^3$) parameters per rule, while a product of 5 two-state models would have just 40 ($= 5 \cdot 2^3$).

Comparison of Inference Algorithms

In our first experiment, we test the relative performance of the various approximate inference methods at both train and test time. In order to include exact inference, we necessarily need to look at a smaller scale example for which exact inference is still feasible. We examined development performance for training and inference on a small product of two parsers, each with two latent states per symbol.

During training, we have several options. We can use exact training by parsing with the fully articulated product of both grammars, or, we can instead use EP, ADF, or independent training. At test time, we can parse using the full product of both grammars, or, we can instead use EP, ADF, or we can use the method of Petrov (2010) wherein we multiply the parsers together in an *ad hoc* fashion.

The results are in Table 5.1. The best reported score, unsurprisingly, is for using exact training and parsing, but using EP for training and parsing results in a relatively small loss of 0.3 F1. ADF, however, suffers a loss of 0.6 F1 over Exact when used for training and parsing. Otherwise, Exact and EP seem to perform fairly similarly at parse time for all training conditions.

In general, there seems to be a gain for using the same method for training and testing. Each testing method performs at its best when using models trained with the same method. Moreover, except for ADF, the converse holds true: the grammars trained with a given parsing method are best decoded using the same method. This result is similar to that found in **stoyanov2012minimum**

Oddly, using Petrov (2010)'s method does not seem to work well at all for jointly trained models, except for ADF. Similarly, joint parsing underperforms Petrov (2010)'s method when using independently trained models. Likely, the joint parsing algorithms are miscalibrating the redundant information present in the two independently-trained models, while the two jointly-trained components come to depend on each other. In fact, the F1 scores for the two separate models of the EP parser are in the 60's.

As expected, ADF does not perform as well as EP. Therefore, we exclude it from our subsequent experiments, focusing exclusively on EP.

Latent Variable Experiments

Most of the previous work in latent variable parsing has focused on splitting smaller unstructured refinements into larger unstructured refinements. Here, we consider training a joint model consisting of a large number of disjoint one-bit (i.e. two-state) latent variable refinements. Specifically, we consider the performance of products of up to 8 one-bit refinements.

In Figure 5.4, we show development F1 as a function of the number of latent bits. Improvement is roughly linear up to 3 components. Performance levels off afterwards, with the top performing system scoring 89.7 F1. Nevertheless, these parsers outperform the comparable parsers of Petrov and Klein (2008a) (89.3), even though our six-bit parser has many fewer effective parameters per binary rule: 48 instead of the 4096 in their best parser. We also ran our best system on Section 23, where it gets 89.1 and 88.4 on sentences less than length 40 and on all sentences, respectively. This result compares favorably to the 88.8/88.3 of Petrov and Klein (2008a).

Runtime Experiments

We also examined how long exact inference and expectation propagation took as we increased the number of components. We plotted runtime, in sentences per second, for each algorithm as a function of the number of latent bits (like in the previous section), calculated on the

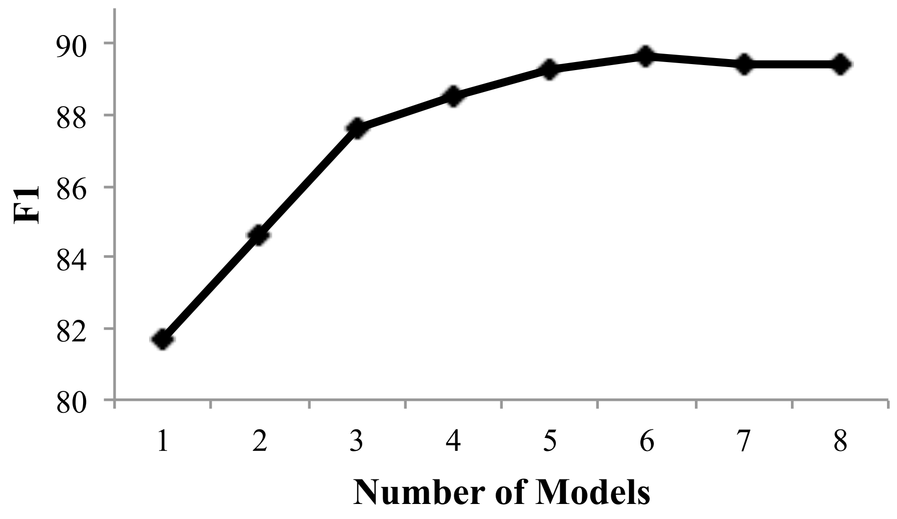


Figure 5.4: Development F1 plotted against the number M of one-bit latent refinement components. The best grammar has 6 one-bit refinements, with 89.7 F1.

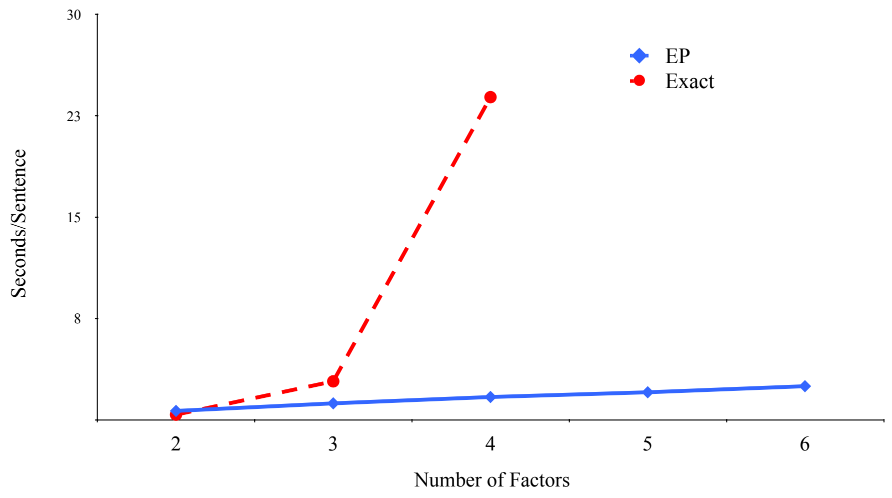


Figure 5.5: Number of seconds to parse a sentence for a given number of factored latent bits. The time taken by exact inference grows exponentially, becoming intractable after just three latent bits. Expectation propagation, on the other hand, shows linear growth.

Models	F1, ≤ 40	F1, All
Lexicalized	87.3	86.5
Structural	86.3	85.4
3xLatent	88.6	87.6
Lex+Struct	90.2	89.5
Lex+Lat	90.0	89.4
Struct+Lat	90.0	89.4
Lex+Struct+Lat	90.2	89.7

Table 5.2: Development F1 score for various model combinations for sentences less than length 40 and all sentences. 3xLatent refers to a latent refinement model with 3 factored latent bits.

	Models			
Lat. Bits	ϕ	Lexicalized	Structural	Lex+Struct
0	——	87.3/86.5	86.3/85.4	90.2/89.5
1	81.6/80.6	89.7/89.0	88.1/87.2	90.2/89.5
2	84.6/83.9	89.8/89.2	88.6/87.8	90.1/89.6
3	87.6/86.1	90.0/89.4	88.7/88.0	90.2/89.7
4	88.5/87.6	89.8/89.2	88.8/88.1	90.0/89.3
5	89.2/88.4	89.9/89.2	88.4/87.7	90.1/89.4
6	89.7/88.9	90.1/89.5	88.5/87.7	90.0/89.3

Table 5.3: Extra development set results with more combinations. The values reported here are Section 22 Len 40/All of Section 22. The numbers on the left hand side are the number of factored latent bit refinements used in conjunction with the model above. The ϕ column uses only latent information.

development set of the Penn Treebank. Figure 5.5 shows how the time taken by exact inference increases rapidly with the number of components, while EP shows linear growth.

Heterogeneous Models

We now consider factored models with different kinds of refinements. Specifically, we tested grammars comprising all subsets of {Lexicalized, Structural, Latent}. We used a model with 3 factored bits as our representative of the latent variable class, because it was closest in performance to the other models. Of course, other smaller and larger combinations are possible, but we found this selection to be representative.

The development results are in Table 5.2, with results for more combinations in Table 5.3. Unsurprisingly, adding more kinds of refinements helps for the most part, though the combination of all three components is not much better than a combination of just the lexicalized and unlexicalized models. Indeed, our best systems involved combining the lexicalized model with some other model. This is probably because the lexicalized model can represent very different syntactic relationships than the latent and unlexicalized models, meaning there is more diversity in the joint model's capacity when using combinations involving the lexicalized refinements.

Finally, we ran our best system (the fully combined one) on Section 23 of the Penn Treebank. It scored 90.1/89.4 F1 on length 40 and all sentences respectively, slightly edging out the 90.0/89.3 F1 of Petrov and Klein (2008a). However, it is not quite as good at exact match: 37.7/35.3 vs 40.1/37.7. Note, though, that their parser makes use of span features, which deliver a gain of +0.3/0.2F1 respectively, while ours does not.

5.6 Analysis

Factored representations capture a fundamental linguistic insight: grammatical categories are not monolithic, unanalyzable entities. Instead, they are composed of numerous facets that together govern how categories combine into parse trees.

We have developed a new model for grammars with factored refinements and presented two methods for parsing with these grammars. Our experiments have demonstrated that our approach produces higher performance parsers with many fewer parameters. Moreover, our model works with both latent and explicit refinements, allowing us to combine linguistic knowledge with machine learning.

Chapter 6

Parsing without Refinements

A preliminary version of this chapter appeared as Hall, Durrett, and Klein (2014).

As we saw in Chapter 3, naïve context-free grammars, such as those embodied by standard treebank refinements, do not parse well because their symbols have too little context to constrain their syntactic behavior. For example, *to* PPs frequently attach to verbs and *of* PPs frequently attach to nouns, but a context-free PP symbol can equally well attach to either. Much of the last few decades of parsing research has therefore focused on propagating contextual information from the leaves of the tree to internal nodes. For example, head lexicalization (Eisner 1996; Collins 1997; Charniak 1997), structural refinement (Johnson 1998a; Klein and Manning 2003), and state-splitting (Matsuzaki, Miyao, and Tsujii 2005; Petrov et al. 2006) are all designed to take coarse symbols like PP and decorate them with additional context. The underlying reason that such propagation is even needed is that probabilistic constituency parsers score trees based on local configurations only, and any information that is not threaded through the tree becomes inaccessible to the scoring function. There have been non-local approaches as well, such as tree-substitution parsers (Bod 1993; Sima'an 2000), neural net parsers (Henderson 2003), and rerankers (Collins and Koo 2005; Charniak and Johnson 2005b; Huang 2008). These non-local approaches can actually go even further in enriching the grammar's structural complexity by coupling larger domains in various ways, though their non-locality generally complicates inference.

In this chapter, we instead try to *minimize* the structural complexity of the grammar by moving as much context as possible onto local surface features. We examine the position that grammars should not propagate any information that is available from surface strings, since a discriminative parser can access that information directly. We therefore begin with a minimal grammar and iteratively augment it with rich input features that do not enrich the context-free backbone. Previous work has also used surface features in their parsers, but the focus has been on machine learning methods (Taskar et al. 2004), latent refinements (Petrov and Klein 2008a; Petrov and Klein 2008b), or implementation (Finkel, Kleeman, and Manning 2008).

By contrast, we investigate the extent to which we need a grammar at all. As a thought experiment, consider a parser with no grammar, which functions by independently classifying each span (i, j) of a sentence as an NP, VP, and so on, or *null* if that span is a non-constituent. For example, spans that begin with *the* might tend to be NPs, while spans that end with *of* might tend to be non-constituents. An independent classification approach is actually very viable for part-of-speech tagging (Toutanova et al. 2003), but is problematic for parsing – if nothing else, parsing comes with a structural requirement that the output be a well-formed, nested tree. Our parser uses a minimal context-free backbone grammar to ensure a basic level of structural well-formedness, but relies mostly on features of surface spans to drive accuracy. Formally, our model is a conditional random field (CRF) where the features factor over anchored rules of a small backbone grammar, as shown in Figure 6.1.

Some aspects of the parsing problem, such as the tree constraint, are clearly best captured by a PCFG. Others, such as heaviness effects, are naturally captured using surface information. The open question is whether surface features are adequate for key effects like subcategorization, which have deep definitions but regular surface reflexes (e.g. the preposition selected by a verb will often linearly follow it). Empirically, the answer seems to be yes, and our system produces strong results, e.g. up to 90.5 F1 on English parsing. Our parser is also able to generalize well across languages with little tuning: it achieves state-of-the-art results on multilingual parsing, scoring higher than the best single-parser system from the SPMRL 2013 Shared Task on a range of languages, as well as on the competition's average F1 metric.

One advantage of a system that relies on surface features and a simple grammar is that it is portable not only across languages but also across tasks to an extent. For example, Socher et al. (2013b) demonstrates that sentiment analysis, which is usually approached as a flat classification task, can be viewed as tree-structured. In their work, they propagate real-valued vectors up a tree using neural tensor nets and see gains from their recursive approach. Our parser can be easily adapted to this task by replacing the X-bar grammar over treebank symbols with a grammar over the sentiment values to encode the output variables and then adding n-gram indicators to our feature set to capture the bulk of the lexical effects. When applied to this task, our system generally matches their accuracy overall and is able to outperform it on the overall sentence-level subtask.

6.1 Parsing Model

Recall from Chapter 3 that our model is a conditional random field (Lafferty, McCallum, and Pereira 2001) over trees. As before, we define the probability of a tree T conditioned on a sentence $\mathbf{w}$ as

$$p(T|\mathbf{w}) \propto \exp\left(\theta^{\intercal} \sum_{r \in T} f(r, \mathbf{w})\right) \tag{6.1}$$

where the feature domains r range over the (anchored) rules used in the tree. An anchored rule r is the conjunction of an unanchored grammar rule $\mathrm{rule}(r)$ and the start, stop, and split indexes where that rule is anchored, which we refer to as span(r). It is important to note that the richness of the backbone grammar is reflected in the structure of the trees T, while the features that condition directly on the input enter the equation through the anchoring span(r).

We start with a simple X-bar grammar whose only symbols are NP, NP-bar, VP, and so on. Our base model has no surface features: formally, on each anchored rule r we have only an indicator of the (unanchored) rule identity, rule(r). As we saw in Chapter 3, this grammar does not parse very accurately.

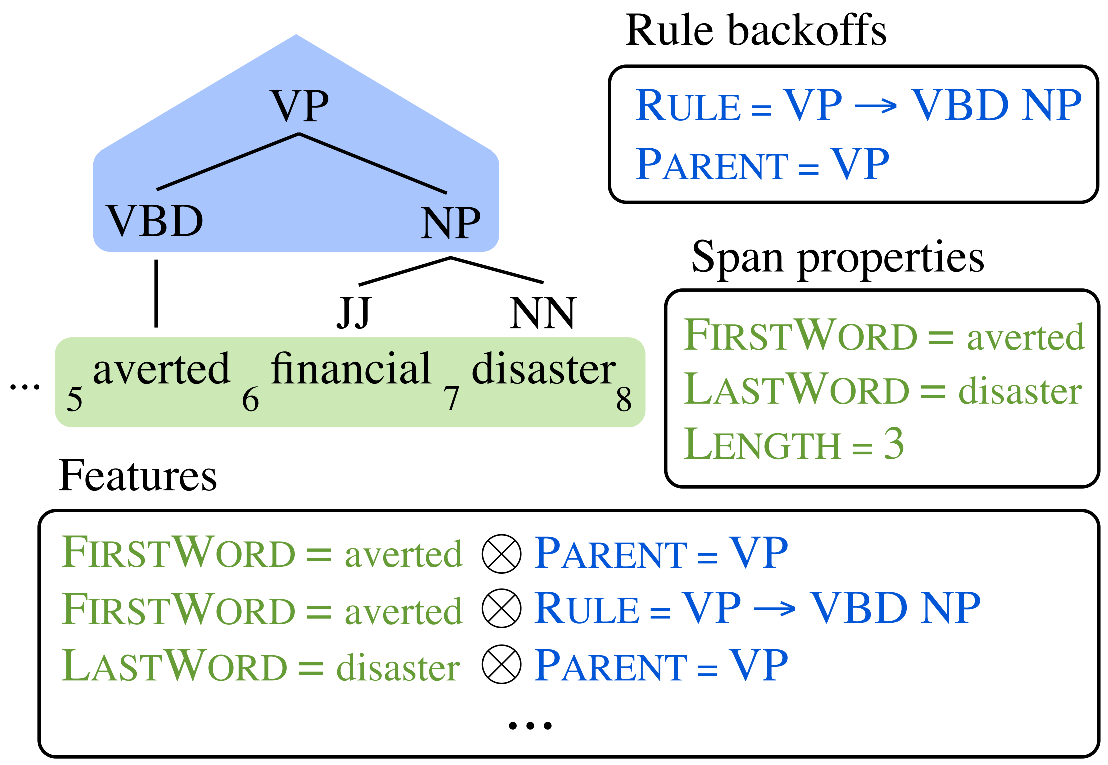
In past work that has used tree-structured CRFs in this way (and in the previous chapters of the thesis), increased accuracy partially came from decorating trees T with additional refinements, giving a tree T' over a more complex symbol set. These refinements introduce additional context into the model, usually capturing linguistic intuition about the factors that influence grammaticality. For instance, we might annotate every constituent X in the tree with its parent Y, giving a tree with symbols $X[\hat{}Y]$. Finkel, Kleeman, and Manning (2008) used parent refinement, head tag refinement, and horizontal sibling refinement together in a single large grammar. In Petrov and Klein (2008a) and Petrov and Klein (2008b), these refinements were latent; they were inferred automatically during training. In the last chapter, we employed both of these kinds of refinements, along with lexicalized head word refinement. All of these CRF parsers (aside from ours in the previous work) do also exploit span features, as did the structured margin parser of Taskar et al. (2004); the current work primarily differs in shifting the work from the grammar to the surface features.

As we saw in the previous chapter, the problem with rich refinements is that they increase the state space of the grammar substantially. For example, adding parent refinement can square the number of symbols, and each subsequent refinement causes a multiplicative increase in the size of the state space. Thus, in the last chapter, we attempted to reduce this state space by factoring these refinements into individual components. The factored approach changed the multiplicative penalty of refinement into an additive penalty, but even so their individual grammar projections are much larger than the base X-bar grammar.

In this chapter, we want to see how much of the expressive capability of refinements can be captured using surface evidence, with little or no refinement of the underlying grammar. To that end, we avoid annotating our trees at all, opting instead to see how far simple surface features will go in achieving a high-performance parser. We will return to the question of refinement in Section 6.5.

6.2 Surface Feature Framework

To improve the performance of our X-bar grammar, we will add a number of surface feature templates derived only from the words in the sentence. We say that an indicator is a surface property if it can be extracted without reference to the parse tree. These features can be implemented without reference to structured linguistic notions like headedness; however, we will argue that they still capture a wide range of linguistic phenomena in a data-driven way.



Rule backoffs
RULE = VP → VBD NP
PARENT = VP
VP
VBD
NP
JJ
NN
... 5 averted 6 financial 7 disaster 8
Span properties
FIRSTWORD = averted
LASTWORD = disaster
LENGTH = 3
Features
FIRSTWORD = averted ⊗ PARENT = VP
FIRSTWORD = averted ⊗ RULE = VP → VBD NP
LASTWORD = disaster ⊗ PARENT = VP
...

Figure 6.1: Features computed over the application of the rule VP → VBD NP over the anchored span *averted financial disaster* with the shown indices. Span properties are generated as described throughout Section 6.3; they are then conjoined with the rule and just the parent nonterminal to give the features fired over the anchored production.

Throughout this and the following section, we will draw on motivating examples from the English Penn Treebank, though similar examples could be equally argued for other languages. For performance on other languages, see Section 6.6.

Recall that our CRF factors over anchored rules r, where each r has identity rule(r) and anchoring span(r). The X-bar grammar has only indicators of rule(r), ignoring the anchoring. Let a *surface property* of r be an indicator function of span(r) and the sentence itself. For example, the first word in a constituent is a surface property, as is the word directly preceding the constituent. As illustrated in Figure 6.1, the actual features of the model are obtained by conjoining surface properties with various abstractions of the rule identity. For rule abstractions, we use two templates: the parent of the rule and the identity of the rule. The surface features are somewhat more involved, and so we introduce them incrementally.

One immediate computational and statistical issue arises from the sheer number of possible surface features. There are a great number of spans in a typical treebank; extracting

Features	Section	F1
Rule	6.3	73.0
+ Span First Word + Span Last Word + Length	6.3	85.0
+ Word Before Span + Word After Span	6.3	89.0
+ Word Before Split + Word After Split	6.3	89.7
+ Span Shape	6.3	89.9

Table 6.1: Results for the Penn Treebank development set, reported in F1 on sentences of length ≤ 40 on Section 22, for a number of incrementally growing feature sets. We show that each feature type presented in Section 6.3 adds benefit over the previous, and in combination they produce a reasonably good yet simple parser.

features for every possible combination of span and rule is prohibitive. One simple solution is to only extract features for rule/span pairs that are actually observed in gold annotated examples during training. Because these "positive" features correspond to observed constituents, they are far less numerous than the set of all possible features extracted from all spans. As far as we can tell, all past CRF parsers have used "positive" features only.

However, negative features—features that are not observed in any tree—are still powerful indicators of (un)grammaticality: if we have never seen a PRN that starts with "has," or a span that begins with a quotation mark and ends with a close bracket, then we would like the model to be able to place negative weights on these features. Thus, we use a simple feature hashing scheme where positive features are indexed individually, while negative features are bucketed together. During training there are no collisions between positive features, which generally receive positive weight, and negative features, which generally receive negative weight; only negative features can collide. Early experiments indicated that using a number of negative buckets equal to the number of positive features was effective.[1]

6.3 Features

Our goal is to use surface features to replicate the functionality of other refinements, without increasing the state space of our grammar, meaning that the rules, rule(r), remain simple, as does the state space used during inference.

Before we present our main features, we briefly discuss the issue of feature sparsity. While lexical features are a powerful driver of our parser, firing features on rare words would allow it to overfit the training data quite heavily. To that end, for the purposes of computing our features, a word is represented by its longest suffix[2] that occurs 100 or more times in the

[1]In Section 6.4, we analyze this choice a little more closely.

[2]We mean "suffix" in the computer science sense, not the linguistic sense. That is, a suffix is any substring that ends at the end of the original string.

training data (which will be the entire word, for common words).[3]

Table 6.1 shows the results of incrementally building up our feature set on the Penn Treebank development set. RULE specifies that we use only indicators on rule identity for binary production and nonterminal unaries. For this experiment and all others, we include a basic set of lexicon features, i.e. features on preterminal part-of-speech tags. A given preterminal unary at position i in the sentence includes features on the words (suffixes) at position $i-1$, i, and $i+1$. Because the lexicon is especially sensitive to morphological effects, we also fire features on all prefixes and suffixes of the current word up to length 5, regardless of frequency.

Subsequent lines in Table 6.1 indicate additional surface feature templates computed over the span, which are then conjoined with the rule identity as shown in Figure 6.1 to give additional features. In the rest of the section, we describe the features of this type that we use. Note that many of these features have been used before (Taskar et al. 2004; Finkel, Kleeman, and Manning 2008; Petrov and Klein 2008b); our goal here is not to amass as many feature templates as possible, but rather to examine the extent to which a simple set of features can replace a complicated state space.

Basic Span Features

We start with some of the most obvious properties available to us, namely, the identity of the first and last words of a span. Because heads of constituents are often at the beginning or the end of a span, these feature templates can (noisily) capture monolexical properties of heads without having to incur the inferential cost of lexicalized refinements. For example, in English, the syntactic head of a verb phrase is typically at the beginning of the span, while the head of a simple noun phrase is the last word. Other languages, like Korean or Japanese, are more consistently head final.

Structural contexts like those captured by parent refinement (Johnson 1998a) are more subtle. Parent refinement can capture, for instance, the difference in distribution in NPs that have S as a parent (that is, subjects) and NPs under VPs (objects). We try to capture some of this same intuition by introducing a feature on the length of a span. For instance, VPs embedded in NPs tend to be short, usually as embedded gerund phrases. Because constituents in the treebank can be quite long, we bin our length features into 8 buckets, of lengths 1, 2, 3, 4, 5, 10, 20, and $\geq$21 words.

Adding these simple features (first word, last word, and lengths) as span features of the X-bar grammar already gives us a substantial improvement over our baseline system, improving the parser's performance from 73.0 F1 to 85.0 F1 (see Table 6.1).

[3]Experiments with the Brown clusters (Brown et al. 1992) provided by Turian, Ratinov, and Bengio (2010) in lieu of suffixes were not promising. Moreover, lowering this threshold did not improve performance.

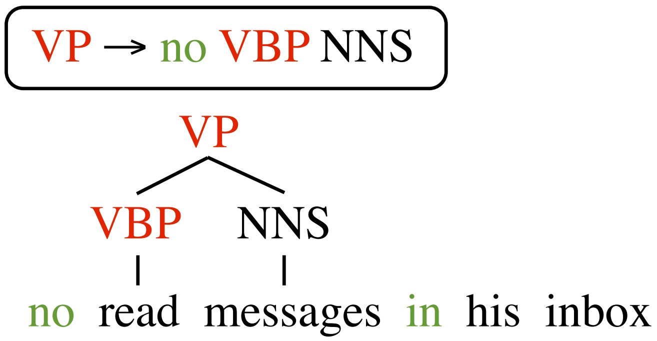


Figure 6.2: An example showing the utility of span context. The ambiguity about whether *read* is an adjective or a verb is resolved when we construct a VP and notice that the word proceeding it is unlikely.

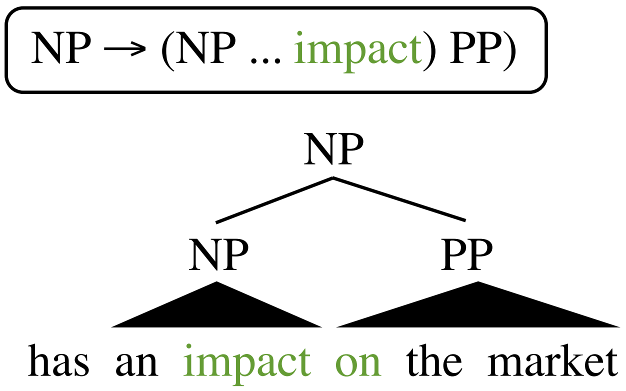


Figure 6.3: An example showing split point features disambiguating a PP attachment. Because *impact* is likely to take a PP, the monolexical indicator feature that conjoins *impact* with the appropriate rule will help us parse this example correctly.

Span Context Features

Of course, there is no reason why we should confine ourselves to just the words within the span: words outside the span also provide a rich source of context. As an example, consider disambiguating the POS tag of the word *read* in Figure 6.2. A VP is most frequently preceded by a subject NP, whose rightmost word is often its head. Therefore, we fire features that (separately) look at the words immediately preceding and immediately following the span.

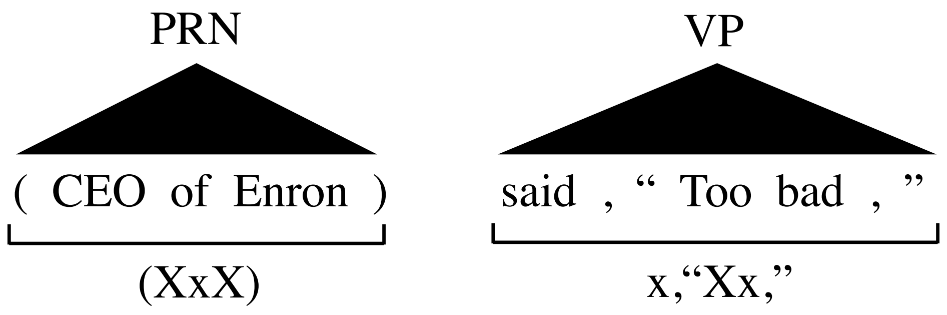


Figure 6.4: Computation of span shape features on two examples. Parentheticals, quotes, and other punctuation-heavy, short constituents benefit from being explicitly modeled by a descriptor like this.

Split Point Features

Another important source of features are the words at and around the split point of a binary rule application. Figure 6.3 shows an example of one instance of this feature template. *impact* is a noun that is more likely to take a PP than other nouns, and so we expect this feature to have high weight and encourage the attachment; this feature proves generally useful in resolving such cases of right-attachments to noun phrases, since the last word of the noun phrase is often the head. As another example, coordination can be represented by an indicator of the conjunction, which comes immediately after the split point. Finally, control structures with infinitival complements can be captured with a rule S $\rightarrow$ NP VP with the word “to” at the split point.

Span Shape Features

We add one final feature characterizing the span, which we call span shape. Figure 6.4 shows how this feature is computed. For each word in the span,[4] we indicate whether that word begins with a capital letter, lowercase letter, digit, or punctuation mark. If it begins with punctuation, we indicate the punctuation mark explicitly. Figure 6.4 shows that this is especially useful in characterizing constructions such as parentheticals and quoted expressions. Because this feature indicates capitalization, it can also capture properties of NP internal structure relevant to named entities, and its sensitivity to capitalization and punctuation makes it useful for recognizing appositive constructions.

[4]For longer spans, we only use words sufficiently close to the span's beginning and end.

Bin Ratio	F1
0.00	88.2
0.01	88.6
0.10	89.9
1.00	90.1
5.00	90.0
10.0	90.1

Table 6.2: Results for the Penn Treebank development set using the same setup as before, varying the number of hash bins for negative features. We examine settings at several multiples of the number of positive features.

6.4 Hash Features

As we mentioned previously, we use feature hashing for "negative" features because there are too many possible features we have not seen to enumerate them all. Using a HyperLogLog sketch (Flajolet et al. 2007), we estimate there are roughly 50 times more negative features than there are positive features if we use an X-bar grammar. If we use refinements—as we do in the next section—it only gets worse, with over a billion features if we use parent annotation. Here we explore parsing accuracy as a function of the number of buckets. Table 6.2 shows parsing performance as a function of the number of hash features (expressed as a multiple of the number of positive features.) For reference, there are roughly 1.3 million positive features.

Clearly, using negative features is quite important: we get gains of up to 1.9F1 from using them. However, once we have "enough" (around 10%) features, performance levels off. We use 100%, as it seemed to work well in a variety of settings.

6.5 Refinements

We have built up a strong set of features by this point, but have not yet answered the question of whether or not grammar refinement is useful on top of them. In this section, we examine two of the most commonly used types of additional refinement, structural refinement, and lexical refinement. Recall from Section 6.2 that every span feature is conjoined with indicators over rules and rule parents to produce features over anchored rule productions; when we consider adding an refinement layer to the grammar, what that does is refine the rule indicators that are conjoined with every span feature. While this is a powerful way of refining features, we show that common successful refinement schemes provide at best modest benefit on top of the base parser.

Refinement	Dev, len ≤ 40
$v = 0, h = 0$	90.1
$v = 1, h = 0$	90.5
$v = 0, h = 1$	90.2
$v = 1, h = 1$	90.9
Lexicalized	90.3

Table 6.3: Results for the Penn Treebank development set, sentences of length ≤ 40, for different refinement schemes implemented on top of the X-bar grammar.

Structural Refinement

The most basic, well-understood kind of refinement on top of an X-bar grammar is structural refinement, which annotates each nonterminal with properties of its environment (Johnson 1998a; Klein and Manning 2003). This includes vertical refinement (parent, grandparent, etc.) as well as horizontal refinement (only partially Markovizing rules as opposed to using an X-bar grammar).

Table 6.3 shows the performance of our feature set in grammars with several different levels of structural refinement.[5] Klein and Manning (2003) find large gains (6% absolute improvement, 20% relative improvement) going from $v = 0, h = 0$ to $v = 1, h = 1$; however, we do not find the same level of benefit. To the extent that our parser needs to make use of extra information in order to apply a rule correctly, simply inspecting the input to determine this information appears to be almost as effective as relying on information threaded through the parser.

In Section 6.6 and Section 6.8, we use $v = 1$ and $h = 0$; we find that $v = 1$ provides a small, reliable improvement across a range of languages and tasks, whereas other refinements are less clearly beneficial.

Lexical Refinement

Another commonly-used kind of structural refinement is lexicalization (Eisner 1996; Collins 1997; Charniak 1997). By annotating grammar nonterminals with their headwords, the idea is to better model phenomena that depend heavily on the semantics of the words involved, such as coordination and PP attachment.

Table 6.3 shows results from lexicalizing the X-bar grammar; it provides meager improvements. One probable reason for this is that our parser already includes monolexical features that inspect the first and last words of each span, which captures the syntactic or the semantic head in many cases or can otherwise provide information about what the constituent's type may be and how it is likely to combine. Lexicalization allows us to capture bilexical

[5]We use $v = 0$ to indicate no refinement, diverging from the notation in Klein and Manning (2003).

	Test ≤ 40	Test all
Berkeley	90.6	90.1
This work	89.9	89.2

Table 6.4: Final Parseval results for the $v = 1, h = 0$ parser on Section 23 of the Penn Treebank.

	Arabic	Basque	French	German	Hebrew	Hungarian	Korean	Polish	Swedish	Avg
	Dev, all lengths									
Berkeley	78.24	69.17	79.74	81.74	87.83	83.90	70.97	84.11	74.50	78.91
Berkeley-Rep	78.70	84.33	79.68	82.74	89.55	89.08	82.84	87.12	75.52	83.28
Our work	78.89	83.74	79.40	83.28	88.06	87.44	81.85	91.10	75.95	83.30
	Test, all lengths									
Berkeley	79.19	70.50	80.38	78.30	86.96	81.62	71.42	79.23	79.18	78.53
Berkeley-Tags	78.66	74.74	79.76	78.28	85.42	85.22	78.56	86.75	80.64	80.89
Our work	78.75	83.39	79.70	78.43	87.18	88.25	80.18	90.66	82.00	83.17

Table 6.5: Results for the nine treebanks in the SPMRL 2013 Shared Task; all values are F-scores for sentences of all lengths using the version of `evalb` distributed with the shared task. Berkeley-Rep is the best single parser from (Björkelund et al. 2013); we only compare to this parser on the development set because neither the system nor test set values are publicly available. Berkeley-Tags is a version of the Berkeley parser run by the task organizers where tags are provided to the model, and is the best single parser submitted to the official task. In both cases, we match or outperform the baseline parsers in aggregate and on the majority of individual languages.

relationships along dependency arcs, but it has been previously shown that these add only marginal benefit to Collins's model anyway (Gildea 2001).

English Evaluation

Finally, Table 6.4 shows our final evaluation on Section 23 of the Penn Treebank. We use the $v = 1, h = 0$ grammar. While we do not do as well as the Berkeley parser, we will see in Section 6.6 that our parser does a substantially better job of generalizing to other languages.

6.6 Other Languages

Historically, many refinement schemes for parsers have required language-specific engineering: for example, lexicalized parsers require a set of head rules and manually-annotated grammars require detailed analysis of the treebank itself (Klein and Manning 2003). A key strength of a parser that does not rely heavily on an annotated grammar is that it may be more portable to other languages. We show that this is indeed the case: on nine languages, our system is competitive with or better than the Berkeley parser, which is the best single parser[6] for the majority of cases we consider.

We evaluate on the constituency treebanks from the Statistical Parsing of Morphologically Rich Languages Shared Task (Seddah et al. 2013). We compare to the Berkeley parser (Petrov and Klein 2007) as well as two variants. First, we use the "Replaced" system of Björkelund et al. (2013) (Berkeley-Rep), which is their best single parser.[7] The "Replaced" system modifies the Berkeley parser by replacing rare words with morphological descriptors of those words computed using language-specific modules, which have been hand-crafted for individual languages or are trained with additional refinement layers in the treebanks that we do not exploit. Unfortunately, Björkelund et al. (2013) only report results on the development set for the Berkeley-Rep model; however, the task organizers also use a version of the Berkeley parser provided with parts of speech from high-quality POS taggers for each language (Berkeley-Tags). These part-of-speech taggers often incorporate substantial knowledge of each language's morphology. Both Berkeley-Rep and Berkeley-Tags make up for some shortcomings of the Berkeley parser's unknown word model, which is tuned to English.

In Table 6.5, we see that our performance is overall substantially higher than that of the Berkeley parser. On the development set, we outperform the Berkeley parser and match the performance of the Berkeley-Rep parser. On the test set, we outperform both the Berkeley parser and the Berkeley-Tags parser on seven of nine languages, losing only on Arabic and French.

These results suggest that the Berkeley parser may be heavily fit to English, particularly in its lexicon. However, even when language-specific unknown word handling is added to the parser, our model still outperforms the Berkeley parser overall, showing that our model generalizes even better across languages than a parser for which this is touted as a strength (Petrov and Klein 2007). Our span features appear to work well on both head-initial and head-final languages (see Basque and Korean in the table), and the fact that our parser performs well on such morphologically-rich languages as Hungarian indicates that our suffix model is sufficient to capture most of the morphological effects relevant to parsing. Of course, a language that was heavily prefixing would likely require this feature to be modified. Likewise, our parser does not perform as well on Arabic and Hebrew. These closely related languages use templatic morphology, for which suffixing is not appropriate; however, using additional surface features based on the output of a morphological analyzer did not lead to increased performance.

Finally, our high performance on languages such as Polish and Swedish, whose training treebanks consist of 6578 and 5000 sentences, respectively, show that our feature-rich model performs robustly even on treebanks much smaller than the Penn Treebank.[8]

[6]I.e. it does not use a reranking step or post-hoc combination of parser results.

[7]Their best parser, and the best overall parser from the shared task, is a reranked product of "Replaced" Berkeley parsers.

[8]The especially strong performance on Polish relative to other systems is partially a result of our model being able to produce unary chains of length two, which occur frequently in the Polish treebank (Björkelund et al. 2013).

6.7 Analysis

One question one might ask is how does this parser fare on longer sentences.[9] We have said that either the first or the last word of a span tend to be the (semantic or syntactic) head of that span. This is especially true for shorter constituents, but not as true for longer constituents. As an example, longer NPs might have modifying adjectival or prepositional phrases, with the head embedded somewhere in the middle of the span. Or, the head of an S is usually the main verb, which is usually somewhere in the middle of the sentence (in English). One possible shortcoming of our approach, then, is that it will not fare as well on longer sentences as approaches that inject information into the grammar. To the extent that the head word is the most "important" word—linguistically speaking—for a constituent, the longer the sentence, the worse this surface feature heuristic becomes.

On the other hand, we use *head outward* binarization for our base grammar. For (unbinarized) rules with a preterminal head child, the head word for the constituent is always on one side or the other of the split point of the binarized rule. In some sense, this choice of binarization along with the split point features mean that we get an approximation of (mono-)lexicalization "for free:" we have a cubic time parsing algorithm that still always has access to the headword.[10] However, we do not have access to bilexical dependencies, but as we mentioned before, Gildea (2001) found that bilexical information did not help in Collins' model, and we found at best modest gains using bilexical features.

In light of the preceding discussion, we were curious to know how our parser performed on sentences of different lengths, relative to a strong baseline. In Figure 6.5, we examine parsing performance (measured in F1) of our system (with and without parent annotation) relative to the parsing performance of the Berkeley Parser for different (binned) lengths of sentences. Specifically, we plotted the ratio of our system's F1 to the Berkeley Parser's on the development set of the Penn Treebank. (We omit the two sentences of length greater than 65.) On the very short sentences (which are few in number), both of our systems are much better than the Berkeley Parser, which actually performs quite poorly. On sentences of moderate length (which make up the bulk of the data), our systems are a little worse than the Berkeley Parser overall, but in general in the same ballpark. On the sentences in the 50-59 word range (of which there are 25), our best system is again better than the Berkeley Parser, while on the four length 60-64 sentences, the Berkeley parser is considerably better, though this could just be noise. So, there is no particularly clear trend in the relative parsing performance of these models, except perhaps for the very long and very short sentences.

[9]Thanks to Julia Hockenmaier for inspiring this line of analysis.

[10]Bilexical constituency parsing has runtime that is $O(n^4)$, or $O(n^5)$ with begin, split, and end point features.

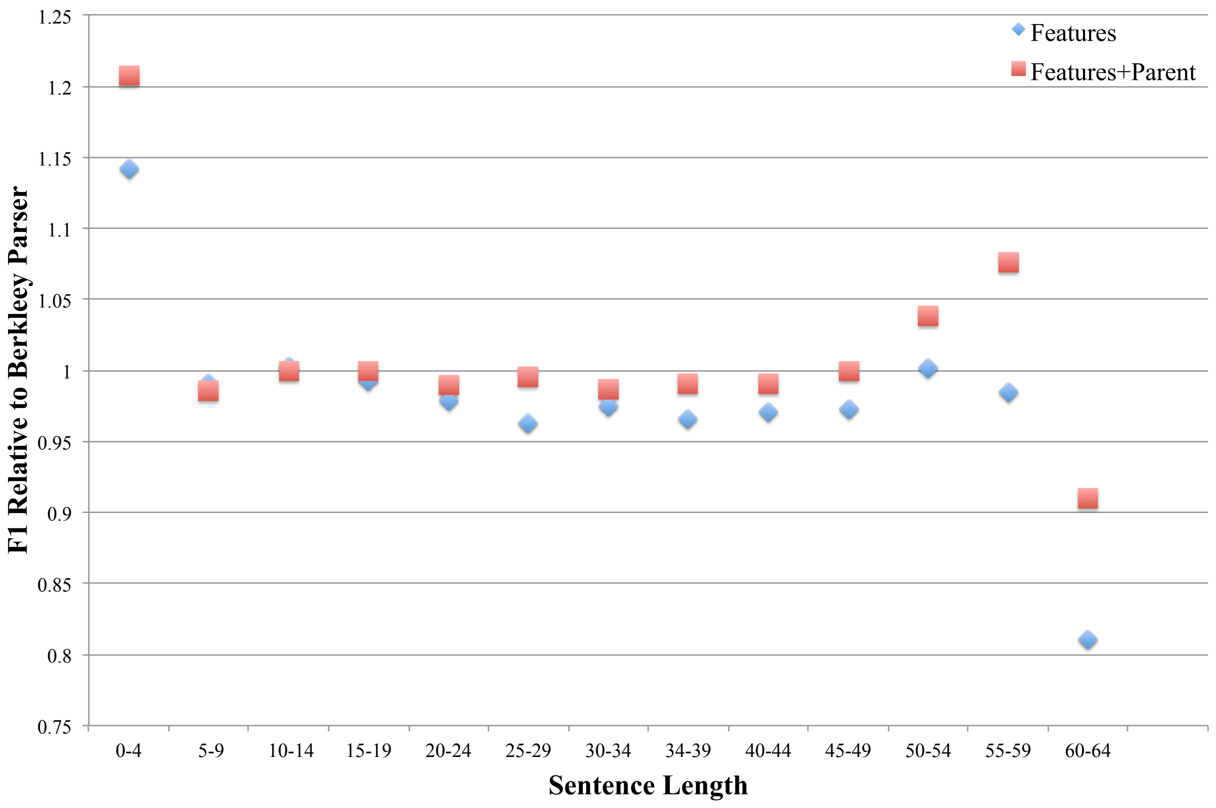


Figure 6.5: Parsing performance compared to the Berkeley Parser's for two of our systems for varying sentence lengths. The Y-axis is the ratio of our systems' F1 to the Berkeley Parser's for sentences of a given length bin. "Features" is our basic system, while "Features+Parent" is our system with parent-annotated grammar.

6.8 Sentiment Analysis

Finally, because the system is, at its core, a classifier of spans, it can be used equally well for tasks that do not normally use parsing algorithms. One example is sentiment analysis. While approaches to sentiment analysis often simply classify the sentence monolithically, treating it as a bag of n-grams (Pang, Lee, and Vaithyanathan 2002; Pang and Lee 2005; Wang and Manning 2012), the recent dataset of Socher et al. (2013b) imposes a layer of structure on the problem that we can exploit. They annotate every constituent in a number of training trees with an integer sentiment value from 1 (very negative) to 5 (very positive), opening the door for models such as ours to learn how syntax can structurally affect sentiment.[11]

Figure 6.6 shows an example that requires some analysis of sentence structure to correctly understand. The first constituent conveys positive sentiment with *never lethargic* and the

[11]Note that the tree structure is assumed to be given; the problem is one of labeling a fixed parse backbone.

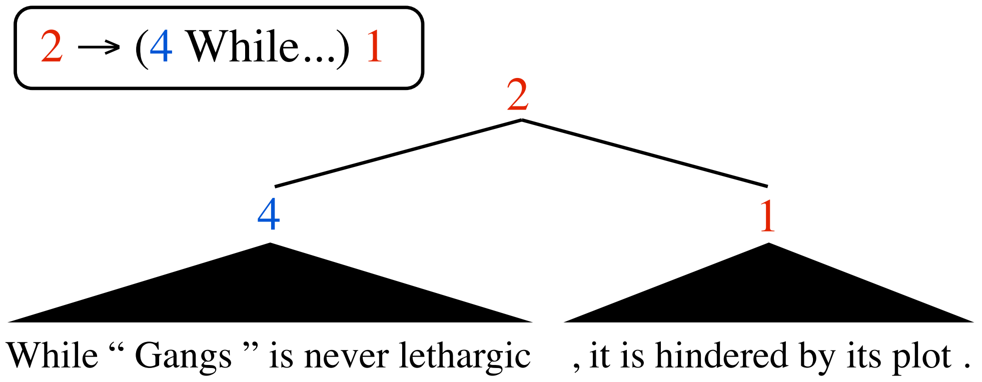

2 → (4 While...) 1
2
4
1
While " Gangs " is never lethargic
, it is hindered by its plot .

Figure 6.6: An example of a sentence from the Stanford Sentiment Treebank which shows the utility of our span features for this task. The presence of "While" under this kind of rule tells us that the sentiment of the constituent to the right dominates the sentiment to the left.

second conveys negative sentiment with *hindered*, but to determine the overall sentiment of the sentence, we need to exploit the fact that *while* signals a discounting of the information that follows it. The grammar rule 2 → 4 1 already encodes the notion of the sentiment of the right child being dominant, so when this is conjoined with our span feature on the first word (*While*), we end up with a feature that captures this effect. Our features can also lexicalize on other discourse connectives such as *but* or *however*, which often occur at the split point between two spans.

Adapting to Sentiment

Our parser is almost entirely unchanged from the parser that we used for syntactic analysis. Though the treebank grammar is substantially different, with the nonterminals consisting of five integers with very different semantics from syntactic nonterminals, we still find that parent refinement is effective and otherwise additional refinement layers are not useful.

One structural difference between sentiment analysis and syntactic parsing lies in where the relevant information is present in a span. Syntax is often driven by heads of constituents, which tend to be located at the beginning or the end, whereas sentiment is more likely to depend on modifiers such as adjectives, which are typically present in the middle of spans. Therefore, we augment our existing model with standard sentiment analysis features that look at unigrams and bigrams in the span (Wang and Manning 2012). Moreover, the Stanford Sentiment Treebank is unique in that each constituent was annotated in isolation, meaning that context never affects sentiment and that every word always has the same tag. We exploit this by adding an additional feature template similar to our span shape feature from Section 6.3 which uses the (deterministic) tag for each word as its descriptor.

	Root	All Spans
Non-neutral Dev (872 trees)		
Stanford CoreNLP current	50.7	80.8
This work	53.1	80.5
Non-neutral Test (1821 trees)		
Stanford CoreNLP current	49.1	80.2
Stanford EMNLP 2013	45.7	80.7
This work	49.6	80.4

Table 6.6: Fine-grained sentiment analysis results on the Stanford Sentiment Treebank of Socher et al. (2013b). We compare against the printed numbers in Socher et al. (2013b) as well as the performance of the corresponding release, namely the sentiment component in the latest version of the Stanford CoreNLP at the time of this writing. Our model handily outperforms the results from Socher et al. (2013b) at root classification and edges out the performance of the latest version of the Stanford system. On all spans of the tree, our model has comparable accuracy to the others.

Results

We evaluated our model on the fine-grained sentiment analysis task presented in Socher et al. (2013b) and compare to their released system. The task is to predict the root sentiment label of each parse tree; however, because the data is annotated with sentiment at each span of each parse tree, we can also evaluate how well our model does at these intermediate computations. Following their experimental conditions, we filter the test set so that it only contains trees with non-neutral sentiment labels at the root.

Table 6.6 shows that our model outperforms the model of Socher et al. (2013b)—both the published numbers and latest released version—on the task of root classification, even though the system was not explicitly designed for this task. Their model has high capacity to model complex interactions of words through a combinatory tensor, but it appears that our simpler, feature-driven model is just as effective at capturing the key effects of compositionality for sentiment analysis.

6.9 Discussion

To date, the most successful constituency parsers have largely been generative, and operate by refining the grammar either manually or automatically so that relevant information is available locally to each parsing decision. Our main contribution in this chapter is to show that there is an alternative to such refinement schemes: namely, conditioning on the input and firing features based on anchored spans. We build up a small set of feature templates as part of a discriminative constituency parser and outperform the Berkeley parser on a wide range of languages. Moreover, we show that our parser is adaptable to other tree-structured tasks such as sentiment analysis; we outperform the recent system of Socher et al. (2013b) and obtain state of the art performance on their dataset.

Chapter 7

Using Refinements to Accelerate GPU Parsing

A preliminary version of this chapter appeared as Hall et al. (2014).

Because NLP models typically treat sentences independently, NLP problems have long been seen as "embarrassingly parallel"—large corpora can be processed arbitrarily fast by simply sending different sentences to different machines. However, recent trends in computer architecture, particularly the development of powerful "general purpose" GPUs, have changed the landscape even for problems that parallelize at the sentence level. First, classic single-core processors and main memory architectures are no longer getting substantially faster over time, so speed gains must now come from parallelism within a single machine. Second, compared to CPUs, GPUs devote a much larger fraction of their computational power to actual arithmetic. Since tasks like parsing boil down to repeated read-multiply-write loops, GPUs should be many times more efficient in time, power, or cost. The challenge is that GPUs are not a good fit for the kinds of sparse computations that most current CPU-based NLP algorithms rely on.

In Canny, Hall, and Klein (2013), we proposed a GPU implementation of a constituency parser that sacrifices all sparsity in exchange for the sheer horsepower that GPUs can provide. That system uses a grammar based on the Berkeley parser (Petrov and Klein 2007) (which is particularly amenable to GPU processing), "compiling" the grammar into a sequence of GPU kernels that are applied densely to every item in the parse chart. Together these kernels implement the Viterbi inside algorithm. On a mid-range GPU, our previous system can compute Viterbi derivations at 164 sentences per second on sentences of length 40 or less (see timing details below).

In this chapter, we develop algorithms that can exploit sparsity on a GPU by adapting coarse-to-fine pruning to a GPU setting. On a CPU, pruning methods can give speedups of up to 100x. Such extreme speedups over a dense GPU baseline currently seem unlikely because fine-grained sparsity appears to be directly at odds with dense parallelism. However, in this chapter, we present a system that finds a middle ground, where some level of sparsity can be

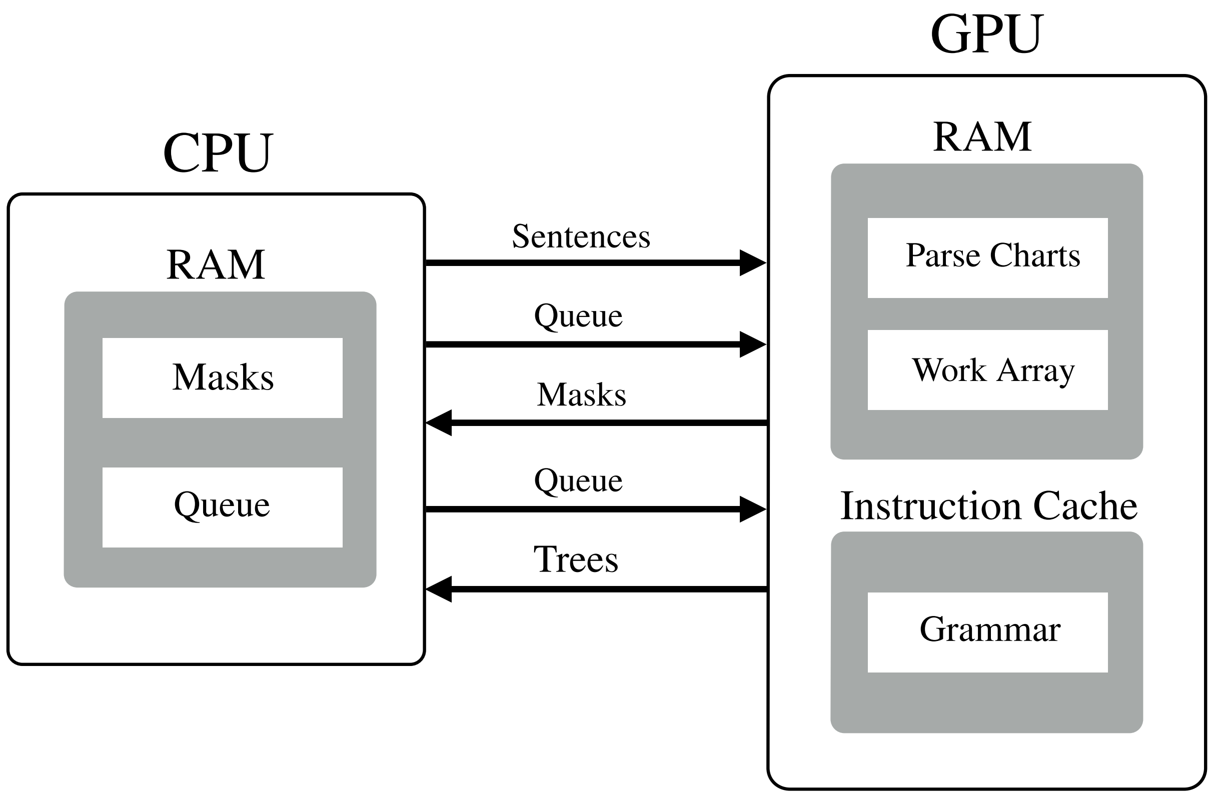


Figure 7.1: Overview of the architecture of our system, which is an extension of Canny, Hall, and Klein (2013)'s system. The GPU and CPU communicate via a work queue, which ferries parse items from the CPU to the GPU. Our system uses a coarse-to-fine approach, where the coarse pass computes a pruning mask that is used by the CPU when deciding which items to queue during the fine pass. The original system of Canny, Hall, and Klein (2013) only used the fine pass, with no pruning.

maintained without losing the parallelism of the GPU. We use a coarse-to-fine approach as in Petrov and Klein (2007), but with only one coarse pass. Figure 7.1 shows an overview of the approach: we first parse densely with a coarse grammar and then parse sparsely with the fine grammar, skipping symbols that the coarse pass deemed sufficiently unlikely. Using this approach, we see a gain of more than 2x over the dense GPU implementation, resulting in overall speeds of up to 404 sentences per second. For comparison, the publicly available CPU implementation of Petrov and Klein (2007) parses approximately 7 sentences per second per core on a modern CPU.

A further drawback of our previous dense approach in Canny, Hall, and Klein (2013) is that it only computes Viterbi parses. As with other grammars with a parse/derivation distinction, the grammars of Petrov and Klein (2007) only achieve their full accuracy using minimum-Bayes-risk parsing, with improvements of over 1.5 F1 over best-derivation Viterbi parsing on the Penn Treebank (Marcus, Santorini, and Marcinkiewicz 1993). To that end, we extend our coarse-to-fine GPU approach to computing marginals, along the way proposing a new way to exploit the coarse pass to avoid expensive log-domain computations in the fine pass. We then implement minimum-Bayes-risk parsing via the max recall algorithm of Goodman (1996). Without the coarse pass, the dense marginal computation is not efficient on a GPU, processing only 32 sentences per second. However, our approach allows us to process over 190 sentences per second, almost a 6x speedup.

7.1 A Note on Experiments

We build up our approach incrementally, with experiments interspersed throughout the chapter, and summarized in Tables 7.1 and 7.2. In this chapter, we focus our attention on current-generation NVIDIA GPUs. Many of the ideas described here apply to other GPUs (such as those from AMD), but some specifics will differ. All experiments are run with an NVIDIA GeForce GTX 680, a mid-range GPU that costs around \$500 at time of writing. Unless otherwise noted, all experiments are conducted on sentences of length ≤ 40 words, and we estimate times based on batches of 20K sentences.[1] We should note that our experimental condition differs from that of Canny, Hall, and Klein (2013): there we evaluated on sentences of length ≤ 30. Furthermore, in that work we used two NVIDIA GeForce GTX *690*s—each of which is essentially a repackaging of two 680s—meaning that our system and experiments would run approximately four times faster on that hardware. (This expected 4x factor is empirically consistent with the result of running that system on our current hardware.)

7.2 Sparsity and CPUs

One successful approach for speeding up constituency parsers has been to use coarse-to-fine inference (Charniak et al. 2006b). In coarse-to-fine inference, we have a sequence of increasingly complex grammars G_ℓ. Typically, each successive grammar G_ℓ is a *refinement* of the preceding grammar $G_{\ell-1}$. That is, for each symbol $A[X]$ in the fine grammar, there is some symbol A in the coarse grammar. For instance, in a latent variable parser, the coarse grammar would have symbols like NP, VP, etc., and the fine pass would have refined symbols NP[0], NP[1], VP[4], and so on.

In coarse-to-fine inference, one applies the grammars in sequence, computing inside and outside scores. Next, one computes (max) marginals for every labeled span (A, i, j) in a sentence. These max marginals are used to compute a *pruning mask* for every span (i, j).

[1]The implementation of Canny, Hall, and Klein (2013) cannot handle batches so large, and so we tested it on batches of 1200 sentences. Our reimplementation is approximately the same speed for the same batch sizes. For batches of 20K sentences, we used sentences from the training set. We verified that there was no significant difference in speed for sentences from the training set and from the test set.

This mask is the set of symbols allowed for that span. Then, in the next pass, one only processes rules that are licensed by the pruning mask computed at the previous level.

This approach works because a low quality coarse grammar can still reliably be used to prune many symbols from the fine chart without loss of accuracy. Petrov and Klein (2007) found that over 98% of symbols can be pruned from typical charts using a simple X-bar grammar without any loss of accuracy. Thus, the vast majority of rules can be skipped, and therefore most computation can be avoided. It is worth pointing out that although 98% of labeled spans can be skipped due to X-bar pruning, we found that only about 79% of binary rule applications can be skipped, because the unpruned symbols tend to be the ones with a larger grammar footprint.

7.3 GPU Architectures

Unfortunately, the standard coarse-to-fine approach does not naïvely translate to GPU architectures. GPUs work by executing thousands of threads at once, but impose the constraint that large blocks of threads must be executing the same instructions in lockstep, differing only in their input data. Thus sparsely skipping rules and symbols will not save any work. Indeed, it may actually slow the system down. In this section, we provide an overview of GPU architectures, focusing on the details that are relevant to building an efficient parser.

The large number of threads that a GPU executes are packaged into blocks of 32 threads called *warps*. All threads in a warp must execute the same instruction at every clock cycle: if one thread takes a branch the others do not, then *all* threads in the warp must follow both code paths. This situation is called *warp divergence*. Because all threads execute all code paths that any thread takes, time can only be saved if an entire warp agrees to skip any particular branch.

NVIDIA GPUs have 8-15 processors called *streaming multi-processors* or SMs.[2] Each SM can process up to 48 different warps at a time: it interleaves the execution of each warp, so that when one warp is stalled another warp can execute. Unlike threads within a single warp, the 48 warps do not have to execute the same instructions. However, the memory architecture is such that they will be faster if they access related memory locations.

A further consideration is that the number of registers available to a thread in a warp is rather limited compared to a CPU. On the 600 series, maximum occupancy can only be achieved if each thread uses at most 63 registers (Nvidia 2008).[3] Registers are many times faster than variables located in thread-local memory, which is actually the same speed as global memory.

[2]Older hardware (600 series or older) has 8 SMs. Newer hardware has more.

[3]A thread can use more registers than this, but the full complement of 48 warps cannot execute if too many are used.

7.4 Anatomy of a Dense GPU Parser

This architecture environment puts very different constraints on parsing algorithms than a CPU environment does. In Canny, Hall, and Klein (2013), we proposed an implementation of a PCFG parser that sacrifices standard sparse methods like coarse-to-fine pruning, focusing instead on maximizing the instruction and memory throughput of the parser. Here, as in our past work, we assume that we are parsing many sentences at once, with throughput being more important than latency. In this section, we describe the dense algorithm, which we take as the baseline for this chapter; we present it in a way that sets up the changes to follow.

At the top level, the CPU and GPU communicate via a *work queue* of parse items of the form (s, i, k, j), where s is an identifier of a sentence, i is the start of a span, k is the split point, and j is the end point. The GPU takes large numbers of parse items and applies the *entire* grammar to them in parallel. These parse items are enqueued in order of increasing span size, blocking until all items of a given length are complete. This approach is diagrammed in Figure 7.2.

Because all rules are applied to all parse items, all threads are executing the same sequence of instructions. Thus, there is no concern of warp divergence.

Grammar Compilation

One important feature of Canny, Hall, and Klein (2013)'s system is *grammar compilation.* Because registers are so much faster than thread-local memory, it is critical to keep as many variables in registers as possible. One way to accomplish this is to unroll loops at compilation time. Therefore, we inlined the iteration over the grammar directly into the GPU kernels (i.e. the code itself), which allows the compiler to more effectively use all of its registers.

However, register space is limited on GPUs. Because the Berkeley grammar is so large, the compiler is not able to efficiently schedule all of the operations in the grammar, resulting in register spills. Previously, we found we had to partition the grammar into multiple different kernels. We discuss this partitioning in more detail in Section 7.6. However, in short, the entire grammar G is broken into multiple clusters G_i where each rule belongs to exactly one cluster.

All in all, the Canny, Hall, and Klein (2013) system is able to compute Viterbi charts at 164 sentences per second, for sentences up to length 40. On larger batch sizes, our new implementation of this approach is able to achieve 193 sentences per second on the same hardware. (See Table 7.1.)

7.5 Pruning on a GPU

Now we turn to the algorithmic and architectural changes in our approach. First, consider trying to directly apply the coarse-to-fine method sketched in Section 7.2 to the dense baseline

Clustering	Pruning	Sent/Sec	Speedup
Canny et al.	–	164.0	–
Reimpl	–	192.9	1.0x
Reimpl	Empty, Coarse	185.5	0.96x
Reimpl	Labeled, Coarse	187.5	0.97x
Parent	–	158.6	0.82x
Parent	Labeled, Coarse	278.9	1.4x
Parent	Labeled, 1-split	**404.7**	**2.1x**
Parent	Labeled, 2-split	343.6	1.8x

Table 7.1: Performance numbers for computing Viterbi inside charts on 20,000 sentences of length ≤40 from the Penn Treebank. All times are measured on an NVIDIA GeForce GTX 680. 'Reimpl' is our reimplementation of our previous approach. Speedups are measured in reference to this reimplementation. See Section 7.6 for discussion of the clustering algorithms and Section 7.5 for a description of the pruning methods. The Canny, Hall, and Klein (2013) system is benchmarked on a batch size of 1200 sentences, the others on 20,000.

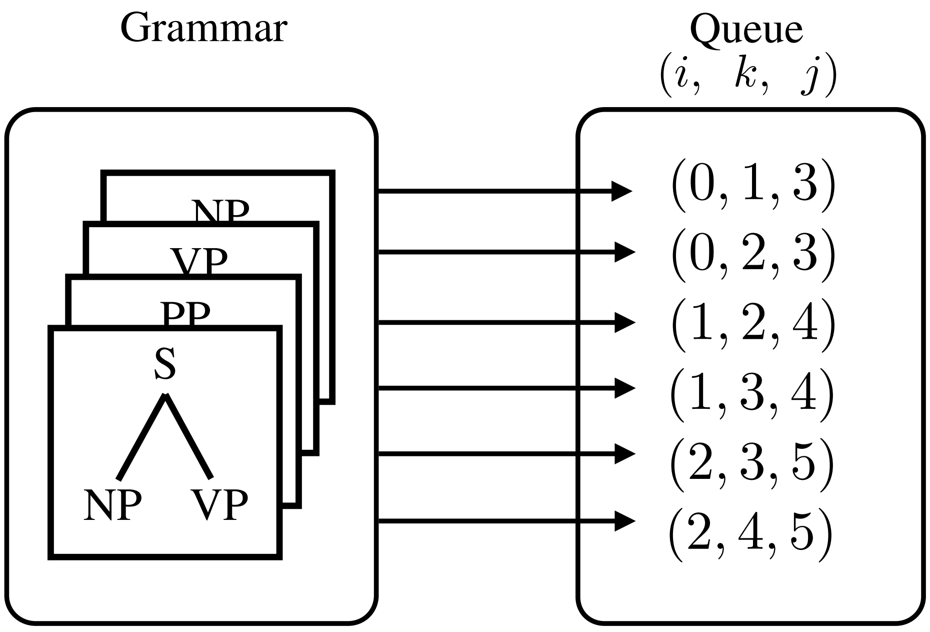


Figure 7.2: Schematic representation of the work queue used in Canny, Hall, and Klein (2013). The Viterbi inside loop for the grammar is inlined into a kernel. The kernel is applied to all items in the queue in a blockwise manner.

described above. The natural implementation would be for each thread to check if each rule is licensed before applying it. However, we would only avoid the work of applying the rule if all threads in the warp agreed to skip it. Since each thread in the warp is processing a different span (perhaps even from a different sentence), consensus from all 32 threads on any

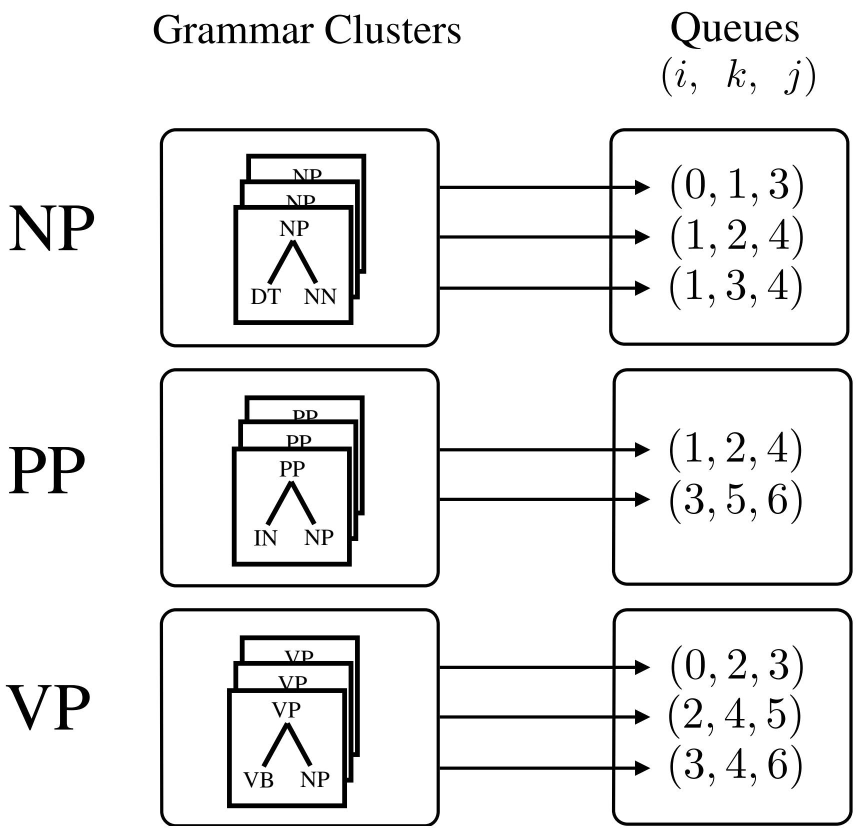


Figure 7.3: Schematic representation of the work queue and grammar clusters used in the fine pass of our work. Here, the rules of the grammar are clustered by their coarse parent symbol. We then have multiple work queues, with parse items only being enqueued if the span (i, j) allows that symbol in its pruning mask.

skip would be unlikely.

Another approach would be to skip enqueuing any parse item (s, i, k, j) where the pruning mask for any of (i, j), (i, k), or (k, j) is entirely empty (i.e. all symbols are pruned in this cell by the coarse grammar). However, our experiments showed that only 40% of parse items are pruned in this manner. Because of the overhead associated with creating pruning masks and the further overhead of GPU communication, we found that this method did not actually produce any time savings at all. The result is a parsing speed of 185.5 sentences per second, as shown in Table 7.1 on the row labeled 'Reimpl' with 'Empty, Coarse' pruning.

Instead, we take advantage of the partitioned structure of the grammar and organize our computation around the coarse symbol set. Recall that the baseline already partitions the grammar G into rule clusters G_i to improve register sharing. (See Section 7.6 for more on the baseline clustering.) We create a separate work queue for each partition. We call each such queue a *labeled work queue*, and each one only queues items to which some rule in the corresponding partition applies. We call the set of coarse symbols for a partition (and therefore the corresponding labeled work queue) a *signature*.

During parsing, we only enqueue items (s, i, k, j) to a labeled queue if two conditions are met. First, the span (i, j)'s pruning mask must have a non-empty intersection with the signature of the queue. Second, the pruning mask for the children (i, k) and (k, j) must be non-empty.

Once on the GPU, parse items are processed using the same style of compiled kernel as in Canny, Hall, and Klein (2013). Because the entire partition (though not necessarily the entire grammar) is applied to each item in the queue, we still do not need to worry about warp divergence.

At the top level, our system first computes pruning masks with a coarse grammar. Then it processes the same sentences with the fine grammar. However, to the extent that the signatures are small, items can be selectively queued only to certain queues. This approach is diagrammed in Figure 7.3.

We tested our new pruning approach using an X-bar grammar as the coarse pass. The resulting speed is 187.5 sentences per second, labeled in Table 7.1 as row labeled 'Reimpl' with 'Labeled, Coarse' pruning. Unfortunately, this approach again does not produce a speedup relative to our reimplemented baseline. To improve upon this result, we need to consider how the grammar clustering interacts with the coarse pruning phase.

7.6 Grammar Clustering

Recall that the rules in the grammar are partitioned into a set of clusters, and that these clusters are further divided into subclusters. How can we best cluster and subcluster the grammar so as to maximize performance? A good clustering will group rules together that use the same symbols, since this means fewer memory accesses to read and write scores for symbols. Moreover, we would like the time spent processing each of the subclusters within a cluster to be about the same. We cannot move on to the next cluster until all threads from a cluster are finished, which means that the time a cluster takes is the amount of time taken by the longest-running subcluster. Finally, when pruning, it is best if symbols that have the same coarse projection are clustered together. That way, we are more likely to be able to skip a subcluster, since fewer distinct symbols need to be "off" for a parse item to be skipped in a given subcluster.

In Canny, Hall, and Klein (2013), we clustered *symbols* of the grammar using a sophisticated spectral clustering algorithm to obtain a permutation of the symbols. Then the rules of the grammar were laid out in a (sparse) three-dimensional tensor, with one dimension representing the parent of the rule, one representing the left child, and one representing the right child. We then split the cube into 6x2x2 contiguous "major cubes," giving a partition of the rules into 24 clusters. We then further subdivided these cubes into 2x2x2 minor cubes, giving 8 subclusters that executed in parallel. Note that the clusters induced by these major and minor cubes need not be of similar sizes; indeed, they often are not. Clustering using this method is labeled 'Reimplementation' in Table 7.1.

The addition of pruning introduces further considerations. First, we have a coarse grammar, with many fewer rules and symbols. Second, we are able to skip a parse item for an entire cluster if that item's pruning mask does not intersect the cluster's signature. Spreading symbols across clusters may be inefficient: if a parse item licenses a given symbol, we will have to enqueue that item to any queue that has the symbol in its signature, no matter how many other symbols are in that cluster.

Thus, it makes sense to choose a clustering algorithm that exploits the structure introduced by the pruning masks. We use a very simple method: we cluster the rules in the grammar by coarse parent symbol. When coarse symbols are extremely unlikely (and therefore have few corresponding rules), we merge their clusters to avoid the overhead of beginning work on clusters where little work has to be done.[4] In order to subcluster, we divide up rules among subclusters so that each subcluster has the same number of active parent symbols. We found this approach to subclustering worked well in practice.

Clustering using this method is labeled 'Parent' in Table 7.1. Now, when we use a coarse pruning pass, we are able to parse nearly 280 sentences per second, a 70% increase in parsing performance relative to Canny, Hall, and Klein (2013)'s system, and nearly 50% over our reimplemented baseline.

It turns out that this simple clustering algorithm produces relatively efficient kernels even in the unpruned case. The unpruned Viterbi computations in a fine grammar using the clustering method of Canny, Hall, and Klein (2013) yields a speed of 193 sentences per second, whereas the same computation using coarse parent clustering has a speed of 159 sentences per second. (See Table 7.1.) This is not as efficient as Canny, Hall, and Klein (2013)'s highly tuned method, but it is still fairly fast, and much simpler to implement.

7.7 Pruning with Finer Grammars

The coarse to fine pruning approach of Petrov and Klein (2007) employs an X-bar grammar as its first pruning phase, but there is no reason why we cannot begin with a more complex grammar for our initial pass. As Petrov and Klein (2007) have shown, intermediate-sized Berkeley grammars prune many more symbols than the X-bar system. However, they are slower to parse with in a CPU context, and so they begin with an X-bar grammar.

Because of the overhead associated with transferring work items to GPU, using a very small grammar may not be an efficient use of the GPU's computational resources. To that end, we tried computing pruning masks with one-split and two-split Berkeley grammars. The X-bar grammar can compute pruning masks at just over 1000 sentences per second, the 1-split grammar parses 858 sentences per second, and the 2-split grammar parses 526 sentences per second.

Because parsing with these grammars is still quite fast, we tried using them as the coarse pass instead. As shown in Table 7.1, using a 1-split grammar as a coarse pass allows us

[4]Specifically, after clustering based on the coarse parent symbol, we merge all clusters with less than 300 rules in them into one large cluster.

to produce over 400 sentences per second, a full 2x improvement over our original system. Conducting a coarse pass with a 2-split grammar is somewhat slower, at a “mere” 343 sentences per second.

7.8 Minimum Bayes risk parsing

The Viterbi algorithm is a reasonably effective method for parsing. However, many authors have noted that parsers benefit substantially from minimum Bayes risk decoding (Goodman 1996; Sima’an 2003; Matsuzaki, Miyao, and Tsujii 2005; Titov and Henderson 2006; Petrov and Klein 2007). MBR algorithms for parsing do not compute the best derivation, as in Viterbi parsing, but instead the parse tree that maximizes the expected count of some figure of merit. For instance, one might want to maximize the expected number of correct constituents (Goodman 1996), or the expected rule counts (Sima’an 2003; Petrov and Klein 2007). MBR parsing has proven especially useful in latent variable grammars. Petrov and Klein (2007) showed that MBR trees substantially improved performance over Viterbi parses for latent variable grammars, earning up to 1.5F1.

Here, we implement the Max Recall algorithm of Goodman (1996). This algorithm maximizes the expected number of correct coarse symbols (A, i, j) with respect to the posterior distribution over parses for a sentence.

This particular MBR algorithm has the advantage that it is relatively straightforward to implement. In essence, we must compute the marginal probability of each fine-labeled span $\mu(A_x, i, j)$, and then marginalize to obtain $\mu(A, i, j)$. Then, for each span (i, j), we find the best possible split point k that maximizes $C(i, j) = \mu(A, i, j) + \max_k (C(i, k) + C(k, j))$. Parse extraction is then just a matter of following back pointers from the root, as in the Viterbi algorithm.[5]

Computing marginal probabilities

The easiest way to compute marginal probabilities is to use the log space semiring rather than the Viterbi semiring, and then to run the inside and outside algorithms as before. We should expect this algorithm to be at least a factor of two slower: the outside pass performs at least as much work as the inside pass. Moreover, it typically has worse memory access patterns, leading to slower performance.

Without pruning, our approach does not handle these log domain computations well at all: we are only able to compute marginals for 32.1 sentences/second, more than a factor of 5 slower than our coarse pass. To begin, log space addition requires significantly more operations than max, which is a primitive operation on GPUs. Beyond the obvious consequence that executing more operations means more time taken, the sheer number of operations becomes too much for the compiler to handle. Because the grammars are compiled into code, the additional operations are all inlined into the kernels, producing much larger kernels.

[5]This is the same algorithm we described in 3.5.

System	Sent/Sec	Speedup
Unpruned Log Sum MBR	32.1	–
Pruned Log Sum MBR	130.4	4.1x
Pruned Scaling MBR	**190.6**	**5.9x**
Pruned Viterbi	404.7	12.6x

Table 7.2: Performance numbers for computing max constituent (Goodman 1996) trees on 20,000 sentences of length 40 or less from the Penn Treebank. For convenience, we have copied our pruned Viterbi system's result.

Indeed, in practice the compiler will often hang if we use the same size grammar clusters as we did for Viterbi. In practice, we found there is an effective maximum of 2000 rules per kernel using log sums, while we can use more than 10,000 rules rules in a single kernel with Viterbi.

With coarse pruning, however, we can avoid much of the increased cost associated with log domain computations. Because so many labeled spans are pruned, we are able to skip many of the grammar clusters and thus avoid many of the expensive operations. Using coarse pruning and log domain calculations, our system produces MBR trees at a rate of 130.4 sentences per second, a four-fold increase.

Scaling with the Coarse Pass

One way to avoid the expense of log domain computations is to use scaled probabilities rather than log probabilities. Scaling is one of the folk techniques that are commonly used in the NLP community, but not generally written about. Recall that floating point numbers are composed of a mantissa m and an exponent e, giving a number $f = m \cdot 2^e$. When a float underflows, the exponent becomes too low to represent in the available number of bits. In scaling, floating point numbers are paired with an additional number that extends the exponent. That is, the number is represented as $f' = f \cdot \exp(s)$. Whenever f becomes either too big or too small, the number is rescaled back to a less "dangerous" range by shifting mass from the exponent e to the scaling factor s.

In practice, one scale s is used for an entire span (i, j), and all scores for that span are rescaled in concert. In our GPU system, multiple scores in any given span are being updated at the same time, which makes this dynamic rescaling tricky and expensive, especially since inter-warp communication is fairly limited.

We propose a much simpler static solution that exploits the coarse pass. In the coarse pass, we compute Viterbi inside and outside scores for every span. Because the grammar used in the coarse pass is a projection of the grammar used in the fine pass, these coarse scores correlate reasonably closely with the probabilities computed in the fine pass: If a span has a very high or very low score in the coarse pass, it typically has a similar score in the fine pass. Thus, we can use the coarse pass's inside and outside scores as the scaling values for the fine pass's scores. That is, in addition to computing a pruning mask, in the coarse pass we store the maximum inside and outside score in each span, giving two arrays of scores $s^I_{i,j}$ and $s^O_{i,j}$. Then, when applying rules in the fine pass, each fine inside score over a split span (i, k, j) is scaled to the appropriate $s^I_{i,j}$ by multiplying the score by $\exp\left(s^I_{i,k} + s^I_{k,j} - s^I_{i,j}\right)$, where $s^I_{i,k}, s^I_{k,j}, s^I_{i,j}$ are the scaling factors for the left child, right child, and parent, respectively. The outside scores are scaled analogously.

By itself, this approach works on nearly every sentence. However, scores for approximately 0.5% of sentences overflow (*sic*). Because we are summing instead of maxing scores in the fine pass, the scaling factors computed using max scores are not quite large enough, and so the rescaled inside probabilities grow too large when multiplied together. Most of this difference arises at the leaves, where the lexicon typically has more uncertainty than higher up in the tree. Therefore, in the fine pass, we normalize the inside scores at the leaves to sum to 1.0.[6] Using this slight modification, no sentences from the Treebank under- or overflow.

We know of no reason why this same trick cannot be employed in more traditional parsers, but it is especially useful here: with this static scaling, we can avoid the costly log sums without introducing any additional inter-thread communication, making the kernels much smaller and much faster. Using scaling, we are able to push our parser to 190.6 sentences/second for MBR extraction, just under half the speed of the Viterbi system.

Parsing Accuracies

It is of course important verify the correctness of our system; one easy way to do so is to examine parsing accuracy, as compared to the original Berkeley parser. We measured parsing accuracy on sentences of length ≤ 40 from section 22 of the Penn Treebank. Our Viterbi parser achieves 89.7 F1, while our MBR parser scores 91.0. These results are nearly identical to the Berkeley parser's most comparable numbers: 89.8 for Viterbi, and 90.9 for their "Max-Rule-Sum" MBR algorithm. These slight differences arise from the usual minor variation in implementation details. In particular, we use one coarse pass instead of several, and a different MBR algorithm. In addition, there are some differences in unary processing.

7.9 Analyzing System Performance

In this section we attempt to break down how exactly our system is spending its time. We do this in an effort to give a sense of how time is spent during computation on GPUs. These timing numbers are computed using the built-in profiling capabilities of the programming environment. As usual, profiles exhibit an observer effect, where the act of measuring the system changes the execution. Nevertheless, the general trends should more or less be preserved as compared to the unprofiled code.

[6]One can instead interpret this approach as changing the scaling factors to $s^{I'}_{i,j} = s^I_{i,j} \cdot \prod_{i \leq k < j} \sum_A \text{inside}(A, k, k+1)$, where inside is the array of scores for the fine pass.

System	Coarse Pass	Fine Pass
Unpruned Viterbi	–	6.4
Pruned Viterbi	1.2	1.5
Unpruned Logsum MBR	—	28.6
Pruned Scaling MBR	1.2	4.3

Table 7.3: Time spent in the passes of our different systems, in seconds per 1000 sentences. Pruning refers to using a 1-split grammar for the coarse pass.

System	Coarse Pass					Fine Pass			
	Binary	Unary	Queueing	Masks	Overhead	Binary	Unary	Queueing	Overhead
Unpruned Viterbi	–	–	–	–	–	5.42	0.14	0.33	0.40
Pruned Viterbi	0.59	0.02	0.19	0.04	0.22	0.56	0.10	0.34	0.22
Pruned Scaling	0.59	0.02	0.19	0.04	0.20	1.74	0.24	0.46	0.84

Table 7.4: Breakdown of time spent in our different systems, in seconds per 1000 sentences. Binary and Unary refer to spent processing binary rules. Queueing refers to the amount of time used to move memory around within the GPU for processing. Overhead includes all other time, which includes communication between the GPU and the CPU.

To begin, we can compute the number of seconds needed to parse 1000 sentences. (We use seconds per sentence rather than sentences per second because the former measure is additive.) The results are in Table 7.3. In the case of pruned Viterbi, pruning reduces the amount of time spent in the fine pass by more than 4x, though half of those gains are lost to computing the pruning masks.

In Table 7.4, we break down the time taken by our system into individual components. As expected, binary rules account for the vast majority of the time in the unpruned Viterbi case, but much less time in the pruned case, with the total time taken for binary rules in the coarse and fine passes taking about 1/5 of the time taken by binaries in the unpruned version. Queueing, which involves copying memory around within the GPU to process the individual parse items, takes a fairly consistent amount of time in all systems. Overhead, which includes transport time between the CPU and GPU and other processing on the CPU, is relatively small for most system configurations. There is greater overhead in the scaling system, because scaling factors are copied to the CPU between the coarse and fine passes.

A final question is: how many sentences per second do we need to process to saturate the GPU's processing power? We computed Viterbi parses of successive powers of 10, from 1 to 100,000 sentences.[7] In Figure 7.4, we then plotted the throughput, in terms of number of sentences per second. Throughput increases through parsing 10,000 sentences, and then levels off by the time it reaches 100,000 sentences.

[7]We replicated the Treebank for the 100,000 sentences pass.

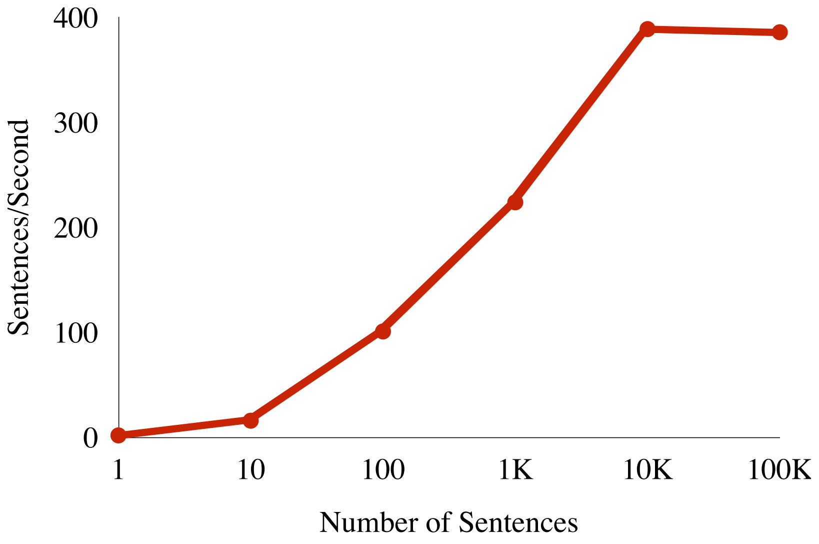


Figure 7.4: Plot of speeds (sentences / second) for various sizes of input corpora. The full power of the GPU parser is only reached when run on large numbers of sentences.

7.10 Related Work

Apart from the model of Canny, Hall, and Klein (2013), there have been a few attempts at using GPUs in NLP contexts before. Johnson (2011) and Yi et al. (2011) both had early attempts at porting parsing algorithms to the GPU. However, they did not demonstrate significantly increased speed over a CPU implementation. In machine translation, He, Lin, and Lopez (2013) adapted algorithms designed for GPUs in the computational biology literature to speed up on-demand phrase table extraction.

7.11 Discussion

GPUs represent a challenging opportunity for natural language processing. By carefully designing within the constraints imposed by the architecture, we have created a parser that can exploit the same kinds of sparsity that have been developed for more traditional architectures.

One of the key remaining challenges going forward is confronting the kind of lexicalized sparsity common in other NLP models. The Berkeley parser's grammars—by virtue of being unlexicalized—can be applied uniformly to all parse items. The bilexical features needed by dependency models and lexicalized constituency models are not directly amenable to acceleration using the techniques we described here. Determining how to efficiently implement these kinds of models is a promising area for new research.

The system described in this chapter is available as open-source at `https://www.github.com/dlwh/puck`.

Chapter 8

Conclusion

In this thesis, we have presented a number of refinements[1] to methods for syntactic parsing. We introduced a unifying framework for talking about different kinds of syntactic refinement, and showed how to combine these different refinements into a larger system in a way that maintains both accuracy and computational tractability. Then, we constructed a parser that eschews refinements in favor of simple surface features. Ultimately, we saw that adding refinements to this model resulted in increased performance, though certain kinds of annotation "stacked" with surface features better than others. Finally, we used refinements to build a new GPU parser that achieved massive speedups relative to CPU systems.

Looking forward, there are a few ways one could extend the work in this thesis. A particularly exciting direction would be to combine the factored approach we took in Chapter 5 with the GPU architecture we described in Chapter 7. One could imagine updating all factors that have identical refinement structure at the same time. A sequence of K two-bit refinements can parallelize at the warp level easily by having each thread in a warp responsible for a different factor for a given cell. Developing this system would require extensions to the core Expectation Propagation algorithm, which usually assumes serial updates. One way to fix this would be to use a convexified version of Expectation Propagation, along the lines described for Belief Propagation in Wainwright, Jaakkola, and Willsky (2003); Wainwright and Jordan (2008b); Schwing et al. (2011). Other possibilities include using different grammatical formalisms in a single factored system (e.g. HPSG, CCG, as well as the standard phrase structure grammar), or to consider less-factored parsers with pairwise interactions between the different factors.

[1]Pun very much intended.

Bibliography

[1] Anne Abeillé, Lionel Clément, and François Toussenel. "Building a treebank for French". In: *Treebanks*. Springer, 2003, pp. 165–187.

[2] I Aldezabal et al. "From dependencies to constituents in the reference corpus for the processing of Basque". In: *Procesamiento del Lenguaje Natural* 41 (2008), pp. 147–154.

[3] Yehoshua Bar-Hillel. "A quasi-arithmetical notation for syntactic description". In: *Language* (1953), pp. 47–58.

[4] Ann Bies et al. "Bracketing guidelines for Treebank II style Penn Treebank project". In: *University of Pennsylvania* 97 (1995).

[5] Sylvie Billott and Bernard Lang. "The Structure of Shared Forests in Ambiguous Parsing". In: *Proceedings of the ACL*. Vancouver, British Columbia, Canada, June 1989, pp. 143–151. DOI: 10.3115/981623.981641. URL: http://www.aclweb.org/anthology/P89-1018.

[6] Anders Björkelund et al. "(Re)ranking Meets Morphosyntax: State-of-the-art Results from the SPMRL 2013 Shared Task". In: *Proceedings of the Fourth Workshop on Statistical Parsing of Morphologically-Rich Languages*. 2013.

[7] Rens Bod. "Using an Annotated Corpus As a Stochastic Grammar". In: *Proceedings of the Sixth Conference on European Chapter of the Association for Computational Linguistics*. 1993.

[8] Xavier Boyen and Daphne Koller. "Tractable Inference for Complex Stochastic Processes". In: *Proceedings of the 14th Conference on Uncertainty in Artificial Intelligence—UAI 1998*. San Francisco: Morgan Kaufmann, 1998, pp. 33–42.

[9] Sabine Brants et al. "The TIGER treebank". In: *Proceedings of the workshop on treebanks and linguistic theories*. Vol. 168. 2002.

[10] Peter F Brown et al. "Class-based n-gram models of natural language". In: *Computational linguistics* 18.4 (1992).

[11] Tim Buckwalter. *ARABIC TRANSLITERATION*. 2002. URL: http://www.qamus.org/transliteration.htm.

[12] David Burkett, John Blitzer, and Dan Klein. "Joint Parsing and Alignment with Weakly Synchronized Grammars". In: *NAACL*. 2010.

[13] David Burkett and Dan Klein. "Fast Inference in Phrase Extraction Models with Belief Propagation". In: *NAACL*. 2012.

[14] John Canny, David Hall, and Dan Klein. "A Multi-Teraflop Constituency Parser using GPUs". In: *Proceedings of EMNLP*. Oct. 2013, pp. 1898–1907. URL: `http://www.aclweb.org/anthology/D13-1195`.

[15] Eugene Charniak. "A Maximum-Entropy-Inspired Parser". In: *1st Meeting of the North American Chapter of the Association for Computational Linguistics*. 2000, pp. 132–139. URL: `http://www.aclweb.org/anthology/A00-2018`.

[16] Eugene Charniak. "Statistical Techniques for Natural Language Parsing". In: *AI Magazine* 18 (1997), pp. 33–44.

[17] Eugene Charniak and Mark Johnson. "Coarse-to-Fine n-Best Parsing and MaxEnt Discriminative Reranking". In: *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*. Ann Arbor, Michigan: Association for Computational Linguistics, 2005, pp. 173–180. URL: `http://www.aclweb.org/anthology/P/P05/P05-1022`.

[18] Eugene Charniak and Mark Johnson. "Coarse-to-fine N-best Parsing and MaxEnt Discriminative Reranking". In: *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*. 2005.

[19] Eugene Charniak et al. "Multilevel coarse-to-fine PCFG parsing". In: *Proceedings of the main conference on Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics*. New York, New York, 2006, pp. 168–175.

[20] Eugene Charniak et al. "Multilevel coarse-to-fine PCFG parsing". In: *Proceedings of the main conference on Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics*. Association for Computational Linguistics. 2006, pp. 168–175.

[21] Key-Sun Choi et al. "KAIST tree bank project for Korean: Present and future development". In: *Proceedings of the International Workshop on Sharable Natural Language Resources*. 1994, pp. 7–14.

[22] N. Chomsky. *Aspects of the theory of syntax*. Cambridge, MA: MIT Press, 1965.

[23] Noam Chomsky. *A minimalist program for linguistic theory*. Vol. 1. MIT, Cambridge Massachusetts: MIT Working Papers in Linguistics, 1992.

[24] Noam Chomsky. "Remarks on nominalization". In: *Reading in English Transformational Grammar*. Ed. by R. Jacobs and P. Rosenbaum. Waltham: Ginn and Co., 1970, pp. 184–221.

[25] Noam Chomsky. *Syntactic Structures*. Mouton, 1957.

[26] Michael Collins. "Three Generative, Lexicalised Models for Statistical Parsing". In: *ACL*. 1997, pp. 16–23.

[27] Michael Collins and Terry Koo. “Discriminative Reranking for Natural Language Parsing”. In: *Computational Linguistics* 31.1 (Mar. 2005), pp. 25–70. ISSN: 0891-2017. DOI: 10.1162/0891201053630273. URL: http://dx.doi.org/10.1162/0891201053630273.

[28] Fabien Cromières and Sadao Kurohashi. “An Alignment Algorithm using Belief Propagation and a Structure-Based Distortion Model”. In: *EACL*. 2009.

[29] Dóra Csendes et al. “The Szeged treebank”. In: *Text, Speech and Dialogue*. Springer. 2005, pp. 123–131.

[30] G. B. Dantzig and P. Wolfe. “Decomposition principle for linear programs”. In: *Operations Research* 8 (1960), pp. 101–111.

[31] Hal Daumé III and Daniel Marcu. “Bayesian Query-Focused Summarization”. In: *Proceedings of the Conference of the Association for Computational Linguistics (ACL)*. Sydney, Australia, 2006. URL: http://pub.hal3.name/#daume06bqfs.

[32] Markus Dreyer and Jason Eisner. “Better Informed Training of Latent Syntactic Features”. In: *EMNLP*. July 2006, pp. 317–326. URL: http://cs.jhu.edu/~jason/papers/#emnlp06.

[33] John Duchi, Elad Hazan, and Yoram Singer. “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization”. In: *COLT* (2010). URL: http://www.colt2010.org/papers/023Duchi.pdf.

[34] Jason Eisner. “Three New Probabilistic Models for Dependency Parsing: An Exploration”. In: *Proceedings of the 16th International Conference on Computational Linguistics (COLING-96)*. 1996.

[35] Jenny Rose Finkel, Alex Kleeman, and Christopher D. Manning. “Efficient, Feature-based, Conditional Random Field Parsing”. In: *ACL 2008*. 2008, pp. 959–967.

[36] Philippe Flajolet et al. “Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm”. In: *IN AOFA ’07: PROCEEDINGS OF THE 2007 INTERNATIONAL CONFERENCE ON ANALYSIS OF ALGORITHMS*. 2007.

[37] Robert W. Floyd. “Algorithm 97: Shortest Path”. In: *Commun. ACM* 5.6 (June 1962), pp. 345–. ISSN: 0001-0782. DOI: 10.1145/367766.368168. URL: http://doi.acm.org/10.1145/367766.368168.

[38] Yoav Freund and Robert E. Schapire. *A Decision-Theoretic Generalization of on-Line Learning and an Application to Boosting*. 1995.

[39] Daniel Gildea. “Corpus Variation and Parser Performance”. In: *Proceedings of Empirical Methods in Natural Language Processing*. 2001.

[40] Joshua Goodman. “Parsing Algorithms and Metrics”. In: *ACL*. 1996, pp. 177–183.

[41] Nizar Habash and Ryan M Roth. “Catib: The columbia arabic treebank”. In: *Proceedings of the ACL-IJCNLP 2009 Conference Short Papers*. Association for Computational Linguistics. 2009, pp. 221–224.

[42] David Hall, Greg Durrett, and Dan Klein. "Less Grammar, More Features". In: *ACL*. 2014.

[43] David Hall and Dan Klein. "Training Factored PCFGs with Expectation Propagation". In: *EMNLP*. 2012.

[44] David Hall et al. "Less Grammar, More Features". In: *ACL*. 2014.

[45] Hua He, Jimmy Lin, and Adam Lopez. "Massively Parallel Suffix Array Queries and On-Demand Phrase Extraction for Statistical Machine Translation Using GPUs". In: *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Atlanta, Georgia: Association for Computational Linguistics, June 2013, pp. 325–334. URL: `http://www.aclweb.org/anthology/N13-1033`.

[46] James Henderson. "Inducing History Representations for Broad Coverage Statistical Parsing". In: *Proceedings of the North American Chapter of the Association for Computational Linguistics on Human Language Technology - Volume 1*. 2003.

[47] Liang Huang. "Forest Reranking: Discriminative Parsing with Non-Local Features". In: *Proceedings of ACL-08: HLT*. Columbus, Ohio: Association for Computational Linguistics, June 2008. URL: `http://www.aclweb.org/anthology/P/P08/P08-1067`.

[48] Ray Jackendoff. *X-bar syntax: a study of phrase structure*. MIT Press, 1977.

[49] Mark Johnson. "Parsing in parallel on multiple cores and gpus". In: *Proceedings of the Australasian Language Technology Association Workshop*. 2011.

[50] Mark Johnson. "PCFG Models of Linguistic Tree Representations". In: *Computational Linguistics* 24.4 (Dec. 1998), pp. 613–632. ISSN: 0891-2017. URL: `http://dl.acm.org/citation.cfm?id=972764.972768`.

[51] Mark Johnson. "PCFG models of linguistic tree representations". In: *Computational Linguistics* 24.4 (1998), pp. 613–632.

[52] Dan Klein and Christopher D. Manning. "Accurate Unlexicalized Parsing." In: *ACL*. 2003, pp. 423–430.

[53] Nikos Komodakis, Nikos Paragios, and Georgios Tziritas. "MRF Optimization via Dual Decomposition: Message-Passing Revisited". In: *ICCV*. 2007, pp. 1–8.

[54] Terry Koo, Xavier Carreras, and Michael Collins. "Simple Semi-supervised Dependency Parsing". In: *Proceedings of ACL-08: HLT*. Columbus, Ohio: Association for Computational Linguistics, June 2008, pp. 595–603. URL: `http://www.aclweb.org/anthology/P/P08/P08-1068`.

[55] Terry Koo et al. "Dual Decomposition for Parsing with Non-Projective Head Automata". In: *Proceedings of EMNLP*. 2010.

[56] John D. Lafferty, Andrew McCallum, and Fernando C. N. Pereira. "Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data". In: *Proceedings of the Eighteenth International Conference on Machine Learning*. 2001.

[57] Mohamed Maamouri et al. “Arabic Treebank: Part 2 v 3.1”. In: *LDC publication 2011T09* (2003).

[58] Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. “Building a Large Annotated Corpus of English: The Penn Treebank”. In: *Computational Linguistics* 19.2 (1993), pp. 313–330.

[59] Takuya Matsuzaki, Yusuke Miyao, and Jun’ichi Tsujii. “Probabilistic CFG with latent annotations”. In: *ACL*. Ann Arbor, Michigan, 2005, pp. 75–82.

[60] Ryan McDonald, Koby Crammer, and Fernando Pereira. “Online large-margin training of dependency parsers”. In: *Proceedings of the 43rd annual meeting on association for computational linguistics*. Association for Computational Linguistics. 2005, pp. 91–98.

[61] Ryan McDonald and Joakim Nivre. “Characterizing the Errors of Data-Driven Dependency Parsing Models”. In: *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*. Prague, Czech Republic: Association for Computational Linguistics, June 2007, pp. 122–131. URL: `http://www.aclweb.org/anthology/D/D07/D07-1013`.

[62] Thomas P. Minka. “Expectation Propagation for approximate Bayesian inference”. In: *UAI*. 2001, pp. 362–369.

[63] Jan Niehues and Stephan Vogel. “Discriminative Word Alignment via Alignment Matrix Modeling”. In: *Proceedings of the Third Workshop on Statistical Machine Translation*. June 2008, pp. 18–25. URL: `http://www.aclweb.org/anthology/W/W08/W08-0303`.

[64] Joakim Nivre, Jens Nilsson, and Johan Hall. “Talbanken05: A Swedish treebank with phrase structure and dependency annotation”. In: *Proceedings of the fifth International Conference on Language Resources and Evaluation (LREC)*. 2006, pp. 1392–1395.

[65] CUDA Nvidia. *Programming guide*. 2008.

[66] Bo Pang and Lillian Lee. “Seeing Stars: Exploiting Class Relationships for Sentiment Categorization with Respect to Rating Scales”. In: *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*. 2005.

[67] Bo Pang, Lillian Lee, and Shivakumar Vaithyanathan. “Thumbs Up?: Sentiment Classification Using Machine Learning Techniques”. In: *Proceedings of the ACL-02 Conference on Empirical Methods in Natural Language Processing - Volume 10*. 2002.

[68] Slav Petrov. “Products of Random Latent Variable Grammars”. In: *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*. Los Angeles, California, June 2010, pp. 19–27. URL: `http://www.aclweb.org/anthology/N10-1003`.

[69] Slav Petrov and Dan Klein. “Discriminative Log-Linear Grammars with Latent Variables”. In: *NIPS*. 2008, pp. 1153–1160. URL: `http://books.nips.cc/papers/files/nips20/NIPS2007_0630.pdf`.

[70] Slav Petrov and Dan Klein. “Improved Inference for Unlexicalized Parsing”. In: *NAACL-HLT*. 2007. URL: `http://www.aclweb.org/anthology/N/N07/N07-1051`.

[71] Slav Petrov and Dan Klein. “Sparse Multi-Scale Grammars for Discriminative Latent Variable Parsing”. In: *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing*. Honolulu, Hawaii: Association for Computational Linguistics, Oct. 2008, pp. 867–876. URL: `http://www.aclweb.org/anthology/D08-1091`.

[72] Slav Petrov et al. “Learning Accurate, Compact, and Interpretable Tree Annotation”. In: *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics*. Sydney, Australia, July 2006, pp. 433–440. URL: `http://www.aclweb.org/anthology/P/P06/P06-1055`.

[73] Carl Pollard and Ivan A. Sag. *Head-Driven Phrase Structure Grammar*. Chicago: University of Chicago Press, 1994.

[74] B. Roy. “Transitivité et connexité”. In: *C. R. Acad. Sci. Paris* 249 (1959), pp. 216–218.

[75] Alexander M Rush et al. “On Dual Decomposition and Linear Programming Relaxations for Natural Language Processing”. In: *EMNLP*. Cambridge, MA, Oct. 2010, pp. 1–11. URL: `http://www.aclweb.org/anthology/D10-1001`.

[76] Lawrence Saul and Michael Jordan. “Exploiting Tractable Substructures in Intractable Networks”. In: *NIPS 1995*. 1996.

[77] Alexander Schwing et al. “Distributed message passing for large scale graphical models”. In: *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*. IEEE. 2011, pp. 1833–1840.

[78] Djamé Seddah et al. “Overview of the SPMRL 2013 Shared Task: A Cross-Framework Evaluation of Parsing Morphologically Rich Languages”. In: *Proceedings of the Fourth Workshop on Statistical Parsing of Morphologically-Rich Languages*. 2013.

[79] Satoshi Sekine and Michael J. Collins. *EVALB – Bracket Scoring Program*. 1997. URL: `http://cs.nyu.edu/cs/projects/proteus/evalb`.

[80] Khalil Sima’an. “Tree-gram Parsing Lexical Dependencies and Structural Relations”. In: *Proceedings of the 38th Annual Meeting on Association for Computational Linguistics*. 2000.

[81] Khalil Sima’an. “On maximizing metrics for syntactic disambiguation”. In: *Proceedings of IWPT*. 2003.

[82] Khalil Sima’an et al. “Building a tree-bank of modern Hebrew text”. In: *Traitement Automatique des Langues* 42.2 (2001), pp. 247–380.

[83] David A. Smith and Jason Eisner. “Dependency Parsing by Belief Propagation”. In: *EMNLP*. Honolulu, Oct. 2008, pp. 145–156. URL: `http://cs.jhu.edu/~jason/papers/#emnlp08-bp`.

[84] Richard Socher et al. “Parsing With Compositional Vector Grammars”. In: *ACL*. 2013.

[85] Richard Socher et al. “Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank”. In: *Proceedings of Empirical Methods in Natural Language Processing*. 2013.

[86] Ben Taskar et al. “Max-Margin Parsing”. In: *In Proceedings of Empirical Methods in Natural Language Processing*. 2004.

[87] Ivan Titov and James Henderson. “Loss minimization in parse reranking”. In: *Proceedings of EMNLP*. Association for Computational Linguistics. 2006, pp. 560–567.

[88] Kristina Toutanova et al. “Feature-rich Part-of-speech Tagging with a Cyclic Dependency Network”. In: *Proceedings of the North American Chapter of the Association for Computational Linguistics on Human Language Technology - Volume 1*. 2003.

[89] Joseph Turian, Lev Ratinov, and Yoshua Bengio. “Word representations: a simple and general method for semi-supervised learning”. In: *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics. 2010.

[90] Martin J Wainwright, Tommi S Jaakkola, and Alan S Willsky. “Tree-reweighted belief propagation algorithms and approximate ML estimation by pseudo-moment matching”. In: *Workshop on Artificial Intelligence and Statistics*. Vol. 21. Society for Artificial Intelligence and Statistics Np. 2003, p. 97.

[91] Martin J Wainwright and Michael I Jordan. *Graphical Models, Exponential Families, and Variational Inference*. Hanover, MA, USA: Now Publishers Inc., 2008. ISBN: 1601981848, 9781601981844.

[92] Martin J Wainwright and Michael I Jordan. “Graphical models, exponential families, and variational inference”. In: *Foundations and Trends® in Machine Learning* 1.1-2 (2008), pp. 1–305.

[93] Sida Wang and Christopher Manning. “Baselines and Bigrams: Simple, Good Sentiment and Topic Classification”. In: *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. 2012.

[94] Stephen Warshall. “A Theorem on Boolean Matrices”. In: *J. ACM* 9.1 (Jan. 1962), pp. 11–12.

[95] Marcin Woliński, Katarzyna Głowińska, and Świdziński Marek. “A preliminary version of Składnica—a treebank of Polish”. In: *Proceedings of the 5th Language & Technology Conference*. 2011.

[96] Eric P. Xing, Michael I. Jordan, and Stuart J. Russell. “A generalized mean field algorithm for variational inference in exponential families”. In: *UAI*. 2003, pp. 583–591.

[97] Youngmin Yi et al. “Efficient Parallel CKY Parsing on GPUs”. In: *Proceedings of the 2011 Conference on Parsing Technologies*. Dublin, Ireland, Oct. 2011.