Formalizing the Use and Characteristics of Constraints in Pipeline Systems

Kristy Hollingshead

B.A., University of Colorado, 2000

M.S., Oregon Health & Science University, 2004

Presented to the Center for Spoken Language Understanding within
the Department of Science & Engineering
and the Oregon Health & Science University
School of Medicine
in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy
in
Computer Science & Engineering

September 2010

Department of Science & Engineering
School of Medicine
Oregon Health & Science University

CERTIFICATE OF APPROVAL

This is to certify that the Ph.D. dissertation of
Kristy Hollingshead
has been approved.

Brian Roark, Thesis Advisor
Associate Professor

Peter Heeman
Associate Research Professor

Deniz Erdogmus
Assistant Professor

Mark Johnson
Professor
Macquarie University

Dedication

To my mom, who has been my cheerleader, sounding board, counselor, and ray of sunshine. She is the reason that this document is before you.

Acknowledgments

First and foremost, I would like to thank my advisor, Dr. Brian Roark, for sharing his expertise and insights with me, and for providing opportunities to pursue my own interests. Thanks are also due to my other committee members, Drs. Peter Heeman, Deniz Erdogmus, and Mark Johnson, for their support of and interest in this work.

I had the great fortune of participating in the JHU CLSP Summer School in 2004 and Summer Workshop in 2008. The long-lasting connections formed there, most notably with Drs. Sanjeev Khudanpur, Chris Callison-Burch, Jason Eisner, and Martin Jansche, have truly shaped me as a researcher. I highly recommend the experience to anyone in the field of natural language processing.

I would also like to thank my colleagues, including Nate Bodenstab, Yongshun Chen, Rachel Coulston, Aaron Dunlop, Seeger Fisher, Josh Flanders, Rebecca Lunsford, Meg Mitchell, Dr. Christian Monson, Emily Tucker Prud'hommeaux, Helen Ross, and Dr. Fan Yang, for making CSLU such a collaborative place full of friendly faces. The departmental staff of CSLU, particularly Pat Dickerson, Kim Basney, and Sean Farrell, deserve special recognition for all their hard work to make our lives easier.

Many special thanks to the DAGGers: Akiko Amano, Qi Miao, and Dr. Taniya Mishra, for mutual support and accountability while we were each working on our dissertations. I am so thankful to have shared this experience with such lovely ladies.

Finally, I would like to acknowledge my families: my dad for sharing his love of language with me, and for two years of ungodly-early phone calls to encourage me to go and write; my mom for listening and understanding; my sister for believing in me and giving me a shoulder to cry on; and my fiancé Tristan for his constant support and encouragement during the time I devoted to writing up this dissertation. I would not have made it without their love and support, and they each have my everlasting thanks.

Contents

List of Tables

List of Figures

Abstract

Formalizing the Use and Characteristics of Constraints in Pipeline Systems

Kristy Hollingshead

Doctor of Philosophy
the Center for Spoken Language Understanding within
the Department of Science & Engineering
and the Oregon Health & Science University
School of Medicine

September 2010
Thesis Advisor: Brian Roark

Pipeline systems, in which data is processed in stages with the output of one stage providing input to the next, are ubiquitous in the field of natural language processing (NLP) as well as many other research areas. The popularity of the pipeline system architecture is due to the utility of pipelines in reducing search complexity, increasing efficiency, and re-using system components. Despite the widespread use of pipelines, there has been little effort toward understanding their functionality and establishing a set of best practices for defining and improving these systems. Improvement techniques are usually discovered in an ad-hoc manner, difficult to generalize for implementation in other systems, and lacking in thorough systematic evaluation of the effects of the technique on the pipeline system.

This dissertation identifies and generalizes shared aspects of pipeline systems across several different application areas, including parsing, speech recognition, machine translation, and image classification. A *formal framework* of pipeline systems is defined, based on these shared aspects, and the argument is made that pipeline improvement techniques derived from this framework will be more easily *generalized* for application to other pipeline systems. A systematic and thorough examination of the *characteristics* of constraints used within several different pipelines is conducted to determine the effects of these characteristics on pipeline performance. This dissertation will define quantitative metrics of constraint characteristics including *diversity, regularity, density,* and *peakedness*. Results will demonstrate that 1) current metrics of constraint quality (typically intrinsic one-best and oracle evaluations) are insufficient predictors of pipeline performance, and 2) several quantitative characteristics of the constraints can be systematically altered to affect pipeline performance. The framework, general improvement techniques, and quantitative measures of the search space as provided by this dissertation, are important steps towards improving the comparison, analysis, and understanding of pipeline systems.

Part I

Introduction and Formal Theory

Chapter 1

Introduction

A *pipeline system* consists of two or more stages for processing data, where the output of one stage provides the input of the next. Pipeline systems are based on a filter-and-refine framework. The first stage in the pipeline *filters* a large set of possible solutions by making a first pass over the search space and producing some subset of solutions for the next stage. The next stage in the pipeline then *refines* the input solution space by using a more complex model to search over the space and select a subset of filtered solutions for the next stage. Each stage in the sequence typically uses increasingly-complex—and thus hopefully increasingly accurate—models to analyze the input data.

Pipeline systems are also used in everyday life. Think of an assembly line at a factory, or, more abstractly, your decision-making process. Imagine that you are planning a family vacation, and your goal is to reserve a couple of rooms at a nice, affordable hotel close to some interesting attractions. Do you begin by searching through the entire list of all hotels everywhere in the world, sorting by cost, rating, and nearby attractions? Probably not; there would be too many possible solutions (hotels) to analyze each one at that level of detail. Thus, you probably begin by choosing where in the world you would like to visit, like Europe, then narrow your search down to a particular area such as France and perhaps then down to the city of interest, Paris. Once you have narrowed the search down to a small geographical region, then you might start looking at the hotels in that region and select one (or a few) based on its pricing, other travelers' reviews of the hotel, etc. Your entire decision process might therefore look something like the pipeline presented in Figure 1.1.

Like finding your dream vacation hotel amongst all possible hotels in the world, finding a high-quality solution in a large space often requires a complex model with many parameters for estimating the distribution of true solutions in the space of all possible solutions. However, using such complex models for exact search is a time-consuming process. A coarse model with fewer parameters, such as "find all 5-star hotels," will search over the space more efficiently, but will typically produce lower-quality solutions since the decreased parameters corresponds to a poorer characterization of the search space – the simpler search parameters did not take into account other features of the solution such as location and price. Similarly, an approximate inference search, in which an arbitrarily-complex model examines only part of the solution space, can be very efficient but may also produce lower-quality solutions depending on which part of the space was examined

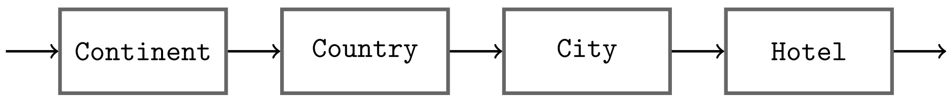


Figure 1.1: Pipeline illustration of an everyday decision-making process.

in the search.

One strategy for defining the space for an approximate inference search is to systematically reduce the search space of the problem in a step-by-step manner in order to apply increasingly complex models to the space without discarding too much of the space at once, thus minimizing the risk of discarding high-quality candidates from the search. The search for a hotel described above is similar, in that first we selected a geographical region, then narrowed that region to a city, and only then did we consider details of the hotels in the area such as ranking and proximity to attractions. Decreasing the size of the search space as complexity increases allows for the application of more complex models which would have been intractable if applied to the entire search space. Reducing the search space in a series of steps is similar to the process of simulated annealing, in which a number of "bad" solution candidates are allowed within the set if inclusion of these candidates allows the search to explore more of the space. As time proceeds, fewer of these low-quality candidates are permitted, which corresponds to reducing the "temperature" in the annealing process, thus reducing the size of the space to be explored. A pipeline architecture provides a skeletal structure for reducing the search space of a problem in a series of stages.

1.1 Pipelines in NLP

One of the reasons that pipeline systems are popular in natural language processing (NLP) is because the search space for many NLP problems is large and multi-dimensional. Finding the global optimum in such a space is computationally very demanding, and thus NLP systems rely on search techniques that *sample* the space such that there is a high probability of finding near-optimal solutions. One of the foundational tasks in NLP is text annotation, which can have a very large search space, as will be shown below. Thus, the pipeline architecture with its ability to reduce search complexity provides an attractive solution for many NLP systems.

1.1.1 Solution Space for NLP Tasks

Annotation tasks in NLP consist of labeling (or *annotating*) natural language text, such as news articles or web content, with linguistic structures. The goal then is to find, for a sequence of words, the correct annotation from amongst all possible annotations. More formally, the task is to find, from amongst the set of all possible annotations $\mathcal{T}$, the most likely annotation $\hat{\mathbf{T}}$ for a word sequence $\mathbf{W}$:

$$\hat{\mathbf{T}} = \operatorname*{argmax}_{\mathbf{T}\in\mathcal{T}} \mathrm{P}(\mathbf{T}|\mathbf{W}) \tag{1.1}$$

Stated another way, the task is to search over the solution space defined by $\mathcal{T}$ to find the best solution according to the model defined by $\mathrm{P}(\mathbf{T}|\mathbf{W})$.

Two common examples of NLP annotation tasks are part-of-speech (POS) tagging and syntactic parsing. The objective of a POS-tagging task is to identify the syntactic function of each word, such as whether a word is a noun or a verb. Similarly, the goal of a parsing task is to identify syntactic constituents such as noun phrases, verb phrases, adjectival phrases, thus identifying sequences of words that tend to function together as a unit. Figure 1.2 shows an example sentence, the POS tags for each word of the sentence, and the syntactic parse of the sentence.

If the word sequence were unambiguous, i.e., corresponding to only one possible annotation, then $|\mathcal{T}|$=1 and search would be simpler. However, this is rarely the case in NLP, because natural language is inherently ambiguous. One example of natural language ambiguity has already been shown: the third word in Figure 1.2(a) ("rose") could function as a noun rather than a verb.

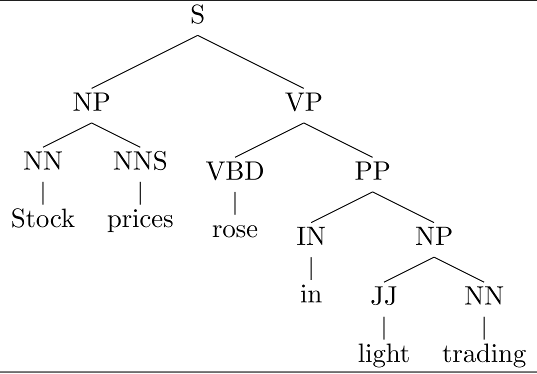
(a)	Stock prices rose in light trading					
(b)	NN	NNS	VBD	IN	JJ	NN
	Stock	prices	rose	in	light	trading
(c)						

Figure 1.2: Example sentence (a) annotated with (b) part-of-speech tags and (c) syntactic parse tree.

In addition to the complexity created by language ambiguity, NLP annotation tasks typically consist of annotating word sequences, rather than single words in isolation. Sequence annotation increases search complexity because the number of possible solutions increases exponentially in the length of the sequence with each additional alternative solution for each word. Consider, again, the POS annotation task. As mentioned above, with two alternative POS tags for just one word in the sequence, there were two possible solutions for the sequence. Our example sentence contains additional ambiguities: "light" could be either an adjective or a noun, and "trading" could be a noun or a gerund verb. With two alternative tags for three words in the sequence, there are eight possible sequence solutions. With three alternative tags for each word in the six-word sequence, a reasonable number of alternatives given the ambiguity of natural language, the number of possible solutions becomes 3^6 ($|\mathcal{T}|$=729). This number becomes computationally unmanageable as we move to longer word sequences, higher levels of ambiguity, and more complex structures.

Furthermore, in working with robust statistical models in NLP, the number of "possible" solutions tends to be larger than just those represented by ambiguity inherent in natural language. A robust statistical model is one which assigns a non-zero probability to a word sequence. However, language is productive, so there are infinitely many possible valid word sequences. If non-zero probabilities must be assigned to all of them, then statistical NLP is more challenging than simply keeping track of which word sequences (and/or which annotations) have been seen and how many times they have been seen: some number of likely sequences will never have been seen at all, even with a plethora of training data. In order to accomplish this goal of creating robust models, most NLP models employ various regularization (or smoothing) techniques [47, 120] which reserve part of the probability mass for unseen words. Since smoothed statistical NLP models will assign non-zero probability to unseen sequences, it could therefore assign non-zero probability to invalid word sequences or annotations, leading to a huge increase in the size of the possible solutions set.

As an example, Figure 1.3 shows ten of the top fifty parse trees as output by a state-of-the-art statistical parser [41] for our example sentence. Note that some of the linguistic ambiguities are apparent in the set while others are not: "light" is tagged as either an adjective (JJ) or a noun (NN), but "rose" is always tagged as a verb (VBD or VBP), never as a noun. The set of trees shown in the figure are a very small sample from the probabilistic distribution assigned by the parsing model, and in fact this model will produce up to 8,190 distinct trees for our example sentence, many more than we might have predicted ourselves, and yet, still less than the infinitely many parses

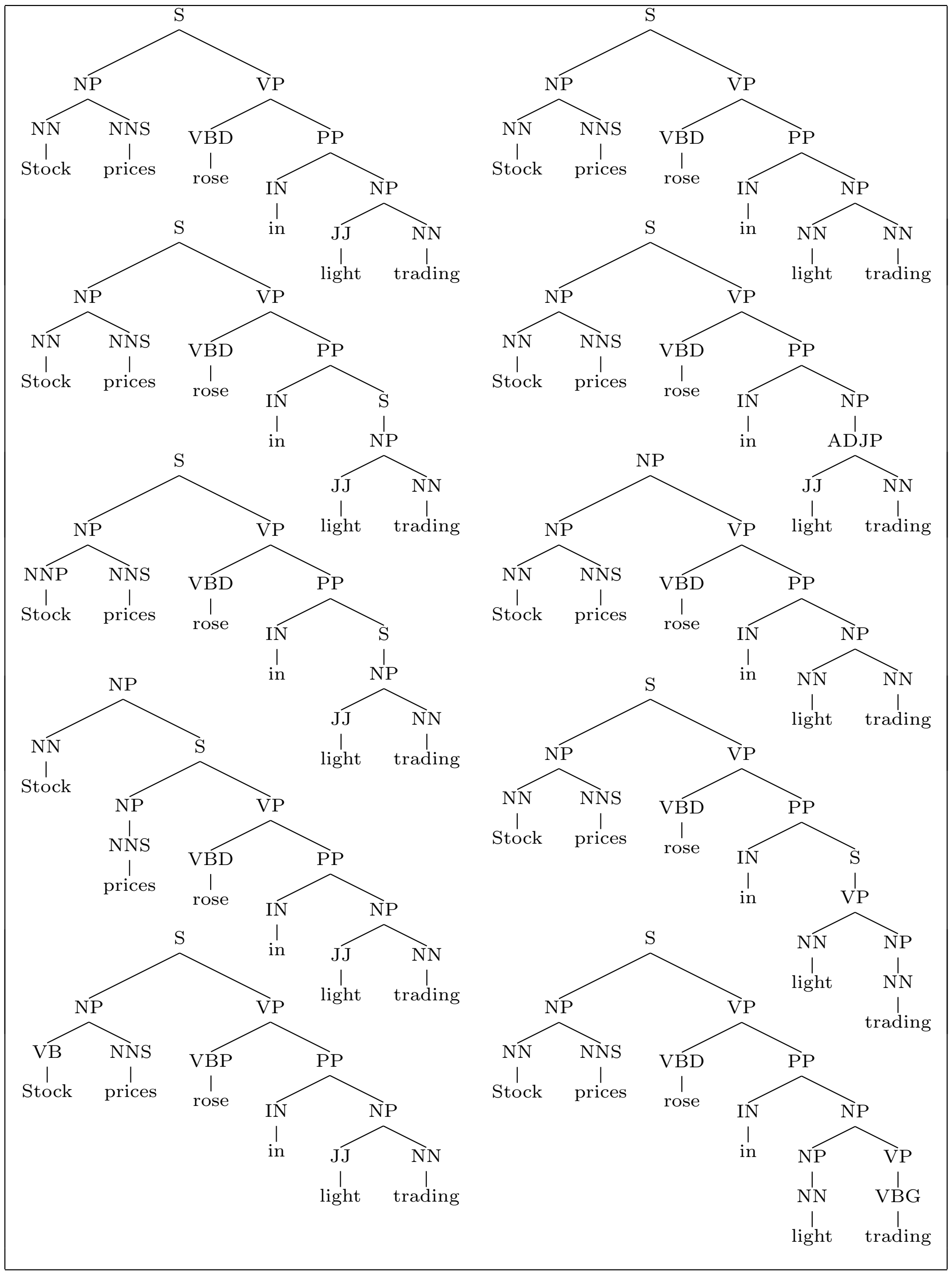


Figure 1.3: Ten of the top fifty parse trees for example sentence.

that might have been predicted under a smoothed exact-inference model.[1] Thus the robustness of NLP statistical models contributes to the overwhelming, computationally unmanageable size of the search space for NLP annotation tasks.

[1] The Charniak [41] parser uses a "coarse-to-fine" strategy which does not explore the entire solution space; see Chapter 3 (p. 56) for more details.

1.1.2 Sequences of Tasks in NLP

The reason that text annotation is such a foundational task in NLP is that one of the purposes of NLP is to aid automated understanding of natural language text, and one approach is by recovering hidden linguistic structures in text, such as syntactic dependencies or semantic relations. Text annotation can aid other NLP tasks, such as machine translation or information retrieval, by providing information about the function each word serves in the text, or the relation between two or more words in a sequence.

For example, in translating our example sentence from English to Spanish, "rose" should be translated as a verb ("subir") rather than a noun ("rosa"), and "light" should be recognized as an adjective in order to correctly re-order "light trading" from the adjective-noun pattern of English to the noun-adjective pattern of Spanish: "intercambio libiano." Thus, annotating word sequences with the syntactic function of the words may be just the first step in a series of annotation tasks.

1.1.3 Filter-and-Refine in NLP

Another way to think about pipelines is within a filter-and-refine framework. Within this framework, each stage necessarily reduces the search space for the next, whereas this reduction is typical but not necessary in a pipeline framework. Under a filter-and-refine framework, the objective of an NLP annotation task becomes one of finding the best annotation $\hat{\mathbf{T}}$ from just the subset of filtered annotations $\mathcal{F}$, requiring a slight alteration to Equation 1.1:

$$\hat{\mathbf{T}} = \operatorname*{argmax}_{\mathbf{T}\in\mathcal{F}\subseteq\mathcal{T}} \mathrm{P}(\mathbf{T}|\mathbf{W}). \tag{1.2}$$

Our search for a vacation hotel falls under the category of a filter-and-refine pipeline, since our task at the final stage was to find the best hotel from amongst just those that passed through our geographical filters in earlier stages. Similarly, the POS-tags in Figure 1.2(b) act as a filter on possible syntax trees for the given sentence. Note in Figure 1.2 that the example syntax tree (c) *contains* the POS-tags (b). Therefore, the input word sequence could be first annotated with POS-tags, then, given the tagged sequence, the syntactic parse could be generated as another layer of annotation. Different POS-tag sequences will often result in different syntactic parses. For example, if the POS-tagger correctly identifies the third word in the example sentence ("rose") as a verb (VBD or VBP), then the parser need not consider parses that place that word in a noun phrase (NP); and in fact, we see in Figure 1.3 that "rose" is always tagged as a verb and thus always a child of a verb phrase (VP). Thus, from the full set of possible parses for the input sentence, the parser need only search over the subset of parse trees that are *consistent with* the POS-tag solution. The POS-tagged sequence identifies a subset of the total space.

In a pipeline system such as the one shown in Figure 1.4, the output from the first-stage tagger *filters out* some of the possible syntactic parse solutions, thus reducing the search space for the second-stage parser. The refinement step in the example pipeline is provided by the parser, at the second stage. One could think of the first-stage POS-tagger as actually providing syntactic parsing solutions, where those solutions are simply the set of parses containing one of the output POS-tag sequences. Of course, the tagger is a greatly-impoverished "parsing model," because it only models the space of possible POS-tag solutions, and does *not* model the higher "layers" of a parse tree. Furthermore, the tagger provides a uniform distribution over the set of consistent parses, so selecting a parse from this set is tantamount to random selection, and it is unlikely that a high-quality parsing solution will be randomly selected. Thus, one can view the parser as *refining* the solution defined by the POS-tagger, with a better model of the remainder of the solution space to improve the likelihood of selecting a higher-quality parse solution.

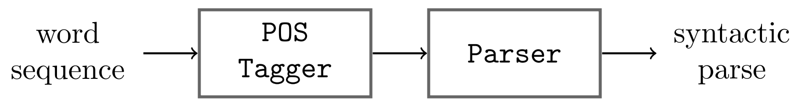


Figure 1.4: An example pipeline system where a sequence of words is input to the first stage, which provides a part-of-speech tagged sequence to the second stage, which produces a syntactic parse tree.

Another, perhaps more straightforward, example of solution refinement is provided by the reranking paradigm, which has recently gained in popularity for several NLP tasks, including parsing [64, 44] and machine translation [199]. In reranking, the solution set output by one pipeline stage is then reranked—or re-sorted—by the next stage, with the goal of using a model with a larger (or different) parameter space in the reranking stage to select a better solution from among the provided solution set. For example, the parser in Figure 1.4 could output the ten highest-scoring parses according to its model, then an additional reranking stage, appended at the end of the pipeline, could be used to select the best parse solution from among those top ten parse candidates.

1.2 Problem Statement

Despite the widespread use of pipelines, there has been no formal definition of them, nor has any effort has been made to understand how they function in order to improve their performance.

The pipeline architecture provides an attractive option for improving the efficiency and tractability of search problems in NLP, as well as providing a structure for building highly modular systems, which allows for the reuse of existing components of the system. Modular systems are further motivated by human processing which appears to follow a modular method [86]. For example, Abney [1] motivates his work on chunking with the intuition that when reading sentences, one typically reads them "a chunk at a time." He argues that the order in which chunks occur is much more flexible than the order of words within chunks, and thus parsing by chunks and then proceeding to attachment has distinct processing advantages, which might help explain why humans utilize a chunk-by-chunk parsing strategy themselves.

Thus for efficiency, tractability, and modularity reasons, pipelines have been implemented for many NLP application areas, including parsing, machine translation, semantic role labeling, and speech recognition. Despite such popular usage of the pipeline architecture, the methods and options for implementing such an architecture have remained mostly unstudied. As a result, most pipeline systems are implemented in an ad-hoc manner, by simply copying the techniques "historically" used for other pipeline systems within the same application area, or worse, without considering previous similar implementations. There has been little discussion of the reasoning behind certain pipeline implementation decisions, how (or whether) such decisions might affect pipeline performance, and how a pipeline might be altered so as to improve its performance. Implementing pipeline systems according to the designs and parameterizations that have worked in the past, without analyzing the reasons behind the success (or failure) of such choices, indicates a lack of understanding of how the pipeline architecture functions, which in turn hinders the definition of effective methods to analyze and improve pipeline systems.

There are, in fact, many commonalities underlying existing pipeline systems, regardless of application area. These commonalities include factors such as how constraints are passed from one stage to the next; whether the pipeline includes a reranking stage or, similarly, a voted recombination stage; whether an intrinsic evaluation can be performed on the output at internal stages of the pipeline, or only extrinsic evaluation on the output of the final pipeline stage. However, the descriptions and definitions of existing pipeline systems are very inconsistent, particularly across different application areas such as parsing and machine translation, making it difficult to compare pipeline systems cross-application. Without a straightforward comparison, it is often difficult to understand how advancements in the use of pipeline systems in one application area might be beneficially applied to other applications. Generalizing pipeline systems by focusing on the commonly-used aspects of the underlying system architecture will ease comparison and allow for the development of general techniques for improving pipeline systems.

1.3 Research Objectives

One of the research objectives is to generalize pipeline systems by the creation of a formal definition and theoretical framework for pipeline systems, then using this framework to classify existing pipeline systems. Another objective is to demonstrate that a more quantitative approach to analyzing pipeline systems may result in a better understanding of—and, consequently, better methods for improving—pipeline systems. Therefore this dissertation will also define a set of metrics to quantify characteristics of the constraints used in pipeline systems, and demonstrate how systematic alterations in those characteristics can affect pipeline performance. The utility of building a better understanding of the effects of the constraints will be demonstrated by establishing several pipeline improvement techniques to systematically alter characteristics of the constraints to improve performance. These techniques will be applied to a class of pipeline systems in order to demonstrate the generality of their results, as opposed to previous ad-hoc and implementation-specific results.

The main question to be addressed by this research is as follows: how is the performance of a pipeline system affected by differences in characteristics of the data output by one pipeline stage as input constraints for the next stage in the pipeline? Of course, such constraints are used quite differently by different pipeline systems, and answers to this question will be affected by how the constraints are used, so the research will begin by addressing the following questions:

1. How is performance affected by different representations of the search space constraints, such as n-best lists versus lattices versus partially-defined solution sets?

2. How is performance affected by the flow of data through the pipeline?

3. How is performance affected by the source of the constraints, including such issues as single versus multiple sources, sampling across the pipeline, and sources outside of the pipeline?

4. How do the effects of the constraints differ in train-time pipelines as compared to test-time pipelines?

These questions will be addressed by creating a framework (Chapter 2) for classifying pipeline systems according to how constraints are used within the pipeline. Such a framework will allow the main research question to be addressed in a generalizable manner, looking specifically at how characteristics of the search space defined by the constraints, such as accuracy (Chapter 6), diversity, regularity, density (Chapter 7), and peakedness (Chapter 8), affect pipeline performance. By addressing these questions, this thesis aims to create a new set of "best practices" for designing, implementing, and testing pipeline systems.

1.4 Thesis Contributions

The contributions of this thesis include: a detailed framework for classifying pipeline systems; new metrics to characterize datasets passed between the stages of a pipeline system, along with an analysis of several different methods used to generate such datasets; and a novel method to configure pipelines. The pipeline framework will identify the different elements of a pipeline system, the different ways in which the elements may be realized, and the possible reasons behind and effects of the different realizations. By defining and utilizing new metrics to characterize the constraints used in a pipeline, this dissertation will provide a better understanding of how constraints affect pipeline performance and how best to alter the stages of and constraints within a pipeline to improve performance. Furthermore, we improve over several state-of-the-art results by implementing a novel pipeline iteration technique, tuning the restrictiveness of input constraints to optimize between precision and recall, altering the density of input constraints, and implementing empirical optimization for probability-score features. We show a 0.6% F-score improvement over the hard baseline defined by the Charniak and Johnson [44] pipeline, using the same pipeline (Chapter 4). We also show new state-of-the-art results for NP-Chunking and shallow parsing, higher than any previously reported result by 0.5% absolute F-score (Chapter 5). We improve the rank-accuracy of n-best parse lists using a distance metric defined in Chapter 7. Most importantly, we discover—and correct for—a few of the reasons for the unexpected and puzzling results previously reported in [110, 152] (Chapters 4, 5, 7, and 8). While these improvements may seem small, recall that we are starting from a hard baseline provided by a state-of-the-art parsing pipeline.

1.5 Thesis Organization

The preceding sections have discussed the different uses and advantages of pipeline systems in NLP, and argued that many of the methods used within a pipeline to represent, characterize, and search over the solution space are generalizable. Having established the general nature of pipelines in NLP, the remaining chapters in this dissertation will construct a framework of pipeline systems, conduct a large-scale survey of existing pipeline systems and classify these systems according to the pipeline framework, establish several general techniques for pipeline improvement based on the different pipeline classes, analyze several characteristics of constraints in pipelines, and systematically alter those characteristics to determine the effects on pipeline performance.

Chapter 2

A Pipeline Framework

Recall that one of the goals of this dissertation is to determine how changes in characteristics of the search space affect pipeline performance. However, differences in the pipelines, such as pipeline design, function, and data representation within the pipeline, might cause pipeline systems to react differently under different conditions. Thus, the objective of this chapter is to define a *framework for classifying pipeline systems*, such that pipelines within the same class tend to exhibit similar characteristics in terms of design, function, and response to changes in the search space. Establishing such a framework for classifying pipeline systems, and then analyzing pipeline performance within each class, will provide a baseline for generalizing which pipeline classes are likely to work best under various different conditions.

2.1 Formal Definitions

Let us formally define a pipeline system as a tuple $\mathcal{P}(S, O, I, C, \Psi)$. Each element in this tuple is defined below.

Pipeline is defined as $\mathcal{P}$, which encapsulates the entire pipeline system, from start to finish, including each stage S_i, constraint set C_i, and passageway Ψ (as defined below, respectively).

Pipeline stages are defined as a set of stages S, $S = \{S_1, \ldots, S_k\}$, which represent the k stages in the pipeline. A stage is defined as a stand-alone system to process data as it passes through the pipeline. For the current stage S_i, S_j for $j < i$ is referred to as an **upstream stage**, and S_j for $j > i$ as a **downstream stage**. The ordered sequence of stages is defined based on the major passageway (see message-passing definition, below) through the pipeline.

Output from each stage is defined as $O = \{O_1, \ldots, O_{|S|}\}$, such that O_i is the output of stage S_i. The output of the final stage in the pipeline, $O_{|S|}$ may be evaluated as the **pipeline-final solution**.

Input to each stage is defined as $I = \{I_1, \ldots, I_{|S|}\}$, again such that I_i is the input to stage S_i. In a pipeline system, the input of downstream stages is typically defined (or constrained) by the output of upstream stages.

Constraints output by each stage are represented by $C = \{C_1, \ldots, C_{|S|}\}$, with one set of constraints per pipeline stage; C_i is the set of constraints from stage S_i. **Constraint characteristics**, which will be thoroughly discussed throughout this dissertation, refer to different characteristics of each constraint set C_i. Note that the constraints C_i may be *identical* to the stage output O_i, or the constraints C_i may be *extracted* from the stage output O_i. Conceptually, output differs from constraints only in that output of a stage is typically evaluated for correctness in an intrinsic evaluation, whereas constraints may not necessarily be evaluated for correctness.

Message-passing is represented as the relation Ψ where the binary relation $\Psi(S_i, S_j)$ indicates the presence of a passageway between stage S_i and stage S_j, and furthermore that the flow of information (or constraints) is passed *from* stage S_i *to* stage S_j. There is typically one identifiable major passageway through the stages of a pipeline, with optional minor or auxiliary branches between other stages.

2.1.1 Motivation

The elements of a pipeline tuple $\mathcal{P}$ defined in the previous section were selected based on extensive research of existing pipeline systems in several NLP application areas: parsing, machine translation, image analysis, and speech recognition. Select parsing pipelines include the state-of-the-art Charniak/Johnson parser/reranker [44], the Clark&Curran combinatory categorical grammar parser [56], and the Ratnaparkhi POS-tagger/NP-chunker/parser pipeline [175]. Machine translation (MT) pipelines include the IBM models for word-alignment [30] as instantiated in the GIZA++ system [164], alignment template models [166], and the state-of-the-art phrase-based models of Pharaoh [128] and Moses [129]. An image segmentation system using a cascade of classifiers to estimate k-nearest neighbors [6] and two voted-combination speech recognition systems [84, 97] round out the set of exemplary pipeline systems that motivated different aspects of the pipeline framework defined in this chapter. Further references will be covered in the next chapter.

The following sections will describe five main aspects of a pipeline $\mathcal{P}(S, O, I, C, \Psi)$: pipeline stages $S_1 \dots S_k$, including stage-internal search algorithms and objectives; different message-passing methods Ψ; and the different types and objectives of constraints $C_1 \dots C_{|S|}$ passed between stages. *Pipeline stages* (Section 2.4) are defined in terms of the necessary conditions for a system to participate in a pipeline, as well as some of the stage-internal design choices that can affect the pipeline. *Message passing* (Section 2.3) is a discussion of the different methods to pass constraint information through a pipeline system, with a focus on the directionality of the information flow. *Constraint representation* (Section 2.2) is a definition of the different methods for representing constraints passed from one stage to the next, as well as how the different methods of representation affect stage-internal design choices.

2.2 Constraint Representations

The first aspect of pipeline systems that we will discuss is the messages, or *constraints* $C_1 \dots C_{|S|}$, passed from one stage of the pipeline to the next. More particularly, this section will address the different methods of representing the constraint data. The two main representation methods are *partial* solutions and *complete* solutions. With partial solutions, the search space defined by the constraints is an under-specified set, and must be built up by the downstream stage while remaining consistent with the defined constraints. With complete solutions, the space defined by the constraints is fully-specified, in that the solutions contained within the space can be enumerated without any knowledge of the downstream stage.

The motivation for examining these two different methods of constraint representation is twofold. First, these representations are sufficient to describe any of the pipeline constraints analyzed in a large-scale survey of pipelines from multiple application areas (Chapter 3). Second, and more importantly for this dissertation, the different constraint representations will tend to exhibit different characteristics and will require different metrics to measure the characteristics of the space. For example, the "diversity" (Section 7.1.1) of a complete solution set is measured differently than the diversity of partial solutions. As another example, an under-specified space may tend to be more "regular" (Section 7.1.2) than a fully-specified search space. Thus, separating pipeline systems into classes with similar behavior based on the constraint representation will prove beneficial for the research goal of determining the effects of constraint characteristics on pipeline performance (Chapter 7).

2.2.1 Partial Solutions

Constraints represented as partial solutions *under-specify* the search space such that, with no knowledge of the next-stage model, one cannot precisely determine the possible output of the stage. Partial-solution constraints leave part of the solution set undefined by placing a restriction on only part of the search space. Rather than exactly defining a set of countable solutions, partial-solutions define a condition for rejecting a solution from inclusion in the set. These constraints may be used to restrict only the unambiguous parts of the solution, explicitly passing off ambiguity resolution to the next stage. The constrained solution parts may also correspond to areas of such high certainty that exploring the area for alternative solutions would be a waste of resources.

An example of a partial solution space is a POS-tag sequence used to constrain a parser: the POS-tag sequence represents a partial parsing solution, and the parser is constrained to just those parses that contain the given POS tags. Of course, one could also use a set of partial solutions to constrain the downstream stage: for example, using several POS-tag sequences to constrain the parser to just those parses that contain any of the given POS-tag sequences. Note that with sets of partial constraints, the downstream stage may interpret the constraints as a union or disjoint set of constraints, each of which will have a slightly different effect on the characteristics of the constraint-defined space.

One of the benefits of partial-solution constraints is that the constraint-defined search space is a more *regular* space, without large gaps in the search space between one possible solution and the next. Complete-solution constraints, discussed in the next section, may hinder performance of downstream stages by producing irregular search spaces. This characteristic of regular versus irregular search spaces will be further explored in Chapter 7.

One of the disadvantages of partial-solution constraints is that the objective of downstream stages will be different than the current stage. The current stage will (in general) optimize for the "best" partial solution, which may or may not correspond to the best full solution. Furthermore, the modularity of pipelines means that the same pipeline stage could be used in several different pipelines, e.g., a POS-tagging stage could be used in a constituent parsing pipeline, a dependency parsing pipeline, a machine translation pipeline, an automatic summarization pipeline, etc. Such a situation creates a separation between the intrinsic and extrinsic evaluations of a pipeline, as discussed in Section 2.5.

2.2.2 Complete Solutions

Constraints represented by complete solutions define an exact specification of the search space. Representing the search space as a set of complete candidates, such as an n-best list, has become a popular choice for pipeline systems in recent years. Part of this popularity may be due to the success of reranking methods such as the Charniak and Johnson [44] and Collins [64, 65] parsing-and-reranking pipelines, but also to the prevalence of efficient algorithms for generating n-best lists [159, 112, 151, 113]. The set of complete solution candidates is passed on to the next stage for ranking, reranking, or recombination, according to the stage-internal search algorithm (Section 2.4).

(a) POS-tagging “Stock prices rose in light trading”

1 (NN Stock) (NNS prices) (VBD rose) (IN in) (JJ light) (NN trading)
2 (NN Stock) (NNS prices) (VBD rose) (IN in) (NN light) (NN trading)
3 (NN Stock) (NNS prices) (VBD rose) (RB in) (JJ light) (NN trading)
4 (NN Stock) (NNS prices) (VBD rose) (RP in) (JJ light) (NN trading)
5 (NN Stock) (NNS prices) (VBD rose) (RBR in) (JJ light) (NN trading)
6 (NN Stock) (NNS prices) (VBD rose) (IN in) (JJ light) (VBG trading)
7 (NN Stock) (NNS prices) (VBD rose) (FW in) (JJ light) (NN trading)
8 (NN Stock) (NNS prices) (VBD rose) (RP in) (NN light) (NN trading)
9 (NN Stock) (NNS prices) (VBD rose) (IN in) (NN light) (VBG trading)
10 (NN Stock) (NNS prices) (VBD rose) (FW in) (NN light) (NN trading)

(b)

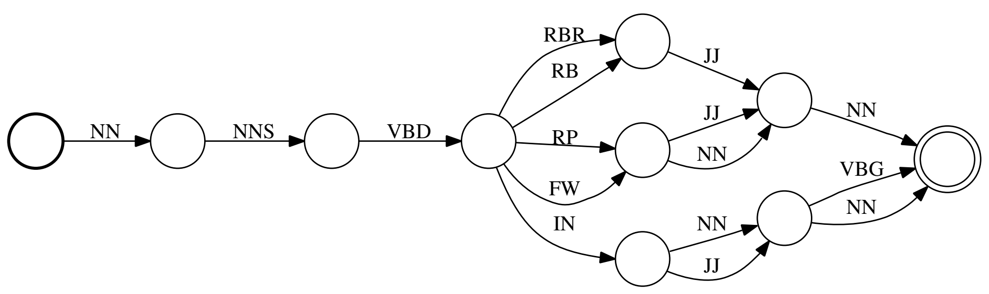


Figure 2.1: Part-of-speech sequence search space as represented by (a) a top-10 list and (b) equivalent graph.

Note that such a set of complete candidates may be represented as a *list* of candidates and also as a *graph* (a word-lattice in ASR, or a CYK chart in parsing). In a candidate-graph, each path through the graph represents one solution candidate. A list can actually be represented as a graph; furthermore, graphs are typically a more compact representation of a list. Figure 2.1(a) shows a list of part-of-speech tags, and Figure 2.1(b) shows the same list in a compacted graph format. Similarly, Figure 2.2(a) shows a list of sentences output by a machine translation decoder while Figure 2.2(b) shows the graph format of the list.

With a full specification of the space, one can exactly enumerate the possible set of solutions to be output with no knowledge of the model used by the next stage. Enumeration of the solutions represented by a list is trivial, of course; enumerating every path through a graph is also straightforward but can require large amounts of memory to do so. This ability to enumerate the space has several benefits.

One of the benefits of representing the search space as a complete solution set is that each solution candidate in the space can be directly analyzed and characterized. For example, the accuracy of each candidate can be calculated (assuming the availability of an evaluation metric for the given data). With these accuracy values, one can then calculate the *oracle* of the space: the solution with the highest accuracy score. The oracle candidate represents the ceiling performance of the next stage downstream, and can be used as the search objective (Section 2.4.2) and in evaluation (Section 2.5).

Another benefit of enumerating the solutions in the space is that it is more straightforward to measure characteristics of the space such as diversity and sparsity. Full specification can appear more restrictive than an under-specification, because the next stage in the pipeline is forced to select from exactly the set of solutions specified. However, it is important to make equal comparisons between two solution spaces. Two fully specified spaces, defined by two n-best lists of possible solutions, are straightforwardly comparable: the smaller n-best list corresponds to more restrictive constraints, since the constrained stage has fewer options from which to select a solution. A partial

(a) Translating “票价格上升，交投清淡”:

1 stock prices rise , trading
2 the stock prices rise , trading
3 in the stock prices rise , trading
4 stock prices rise , trading market
5 the stock price increases , trading
6 trading stock prices rise ,
7 the stock price rise , trading
8 stock prices rose , trading
9 the stock price increases trading
10 stock prices , trading

(b)

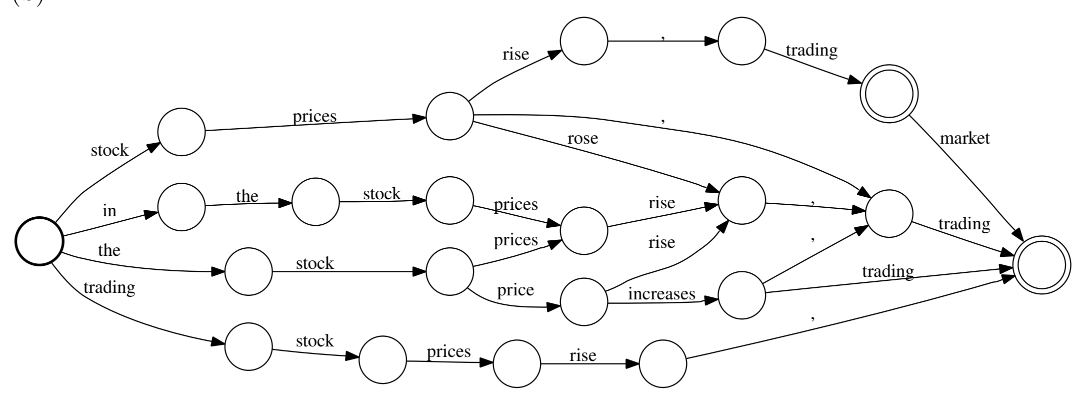


Figure 2.2: Machine translation sequence search space as represented by (a) a top-10 list and (b) equivalent graph.

solution may appear less restrictive, but if the partial solution corresponds to only a few solutions, then this partial-solution constraint is more restrictive than a list of fifty possible solutions. Thus great care should be taken in comparing search spaces defined by partial solutions versus complete solutions.

2.2.3 Hard vs Soft Constraints

The previously mentioned representations of constraints in a pipeline system—both partial- and complete-solution constraints—are examples of *hard* constraints. Hard constraints are used to reduce the size of the search space, thus providing the efficiency benefit of pipeline systems, by systematically reducing the search space at each stage of the pipeline. This class of constraints places an absolute limit on performance capacity; the constrained stage can only select from the solutions included within this limit.

Soft constraints may also be used to define a search space; such constraints do not actually change the search space but provide a prior distribution over the space. Thus the constrained downstream stage is free to select any solution, even ones that were estimated to have little chance of being correct.

These two categories of constraints are not mutually exclusive; a set of constraints could both define a solution subset and provide a probability distribution over that subset, in which case the constraints would be categorized as *both* hard and soft. In fact, most pipeline constraints that we review in Chapter 3 fall under this category. This dissertation will strive to separate the effects of hard versus soft constraints on pipeline performance; in particular, Chapters 6 and 7 examine the effects of hard constraints, while Chapter 8 focuses on the effects of adding soft constraints to a pipelined search space.

2.3 Message-Passing

Message-passing Ψ is the manner in which data is passed through a pipeline system. The messages themselves, and the manner in which they are represented, were discussed in the previous section (2.2). The three methods of passing data through a pipeline system are in a feed-forward manner, feedback manner, or by iteration. This section will also touch on the different possible sources of messages passed through the pipeline.

2.3.1 Feed-forward

In a *feed-forward* pipeline, data continuously flows downstream, from one stage S_i to the next stage S_{i+1} downstream, without circling back upstream. The data makes a single pass through the pipeline. In general, the solution space passed from stage to stage steadily decreases over the course of the pipeline. Figure 2.3 shows a simple feed-forward pipeline, which could be defined as follows: $\mathcal{P}(\{S_1, S_2, S_3\}, \{O_1, O_2, O_3\}, \{I_1, I_2, I_3\}, \{C_1, C_2\}, \{\Psi(S_1, S_2), \Psi(S_2, S_3)\})$, with three stages, constraint sets output from stages 1 and 2, and a message-passing relation from stage 1 to 2, and from stage 2 to 3. Note that there are no passages from downstream stages to upstream stages earlier in the pipeline: $\neg\exists\ \Psi(S_i, S_j)$ s.t. $j \leq i$.

Mathematically, a feed-forward pipeline is equivalent to the *composition* of each of its stages, where each stage in the pipeline corresponds to a function in the composition: $g(f(x))$, or $(g \circ f)(x)$. The range of the inner function of the composition restricts the domain of the composed function; if $f(x) = x^2$, then $g(f(x))$ is restricted to the positive domain and is undefined in the negative domain. Thus in function composition, the range of the inner function filters and reduces the set of possible solutions just as the output of earlier pipeline stages serves to define a reduced search space for later stages.

Our vacation pipeline from Figure 1.1 (p. 2) is an example of a feed-forward pipeline system. Other examples of feed-forward pipelines from NLP include: inputting text to a part-of-speech tagger, then passing the POS output to a downstream parsing stage; inputting parallel corpora text through a word-alignment stage, using the output alignments to extract aligned phrases, then passing the phrases to a downstream machine translation decoding stage; inputting an image to a segmentation stage to identify boundaries in the image, then passing the segmented image to a downstream image labeling stage.

2.3.2 Feedback

In a *feedback* pipeline, data may flow back and forth between pairs of stages. The data makes a single pass through the entire pipeline, but may make multiple passes through any given stage. In general, the solution space passed from stage to stage typically decreases over the course of

$$\begin{tikzcd} {} \arrow[r] & \boxed{S_1} \arrow[r] & \boxed{S_2} \arrow[r] & \boxed{S_3} \arrow[r] & {} \end{tikzcd}$$

Figure 2.3: Feed-forward pipeline.

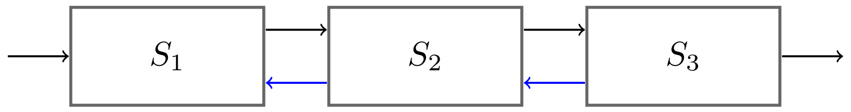


Figure 2.4: Feedback pipeline.

the pipeline, though the feedback mechanism allows for expansion of the space as will be discussed. Figure 2.4 shows a simple feedback pipeline, with bi-directional passages between each pair of stages. This pipeline could be defined as follows: $\mathcal{P}(\{S_1, S_2, S_3\}, \{O_1, O_2, O_3\}, \{I_1, I_2, I_3\}, \{C_1, C_2, C_3\}, \{\Psi(S_1, S_2), \Psi(S_2, S_1), \Psi(S_2, S_3), \Psi(S_3, S_2)\})$, with three stages, message-passing from S_1 to S_2 and back from S_2 to S_1, as well as from stage S_2 to S_3 and back from S_3 to S_2, and constraint sets C_1 from stage S_1, C_2 from S_2, and C_3 from S_3. Note that messages only flow back upstream in passages that already exist for downstream message passing: $\exists\ \Psi(S_j, S_i)$, where $j > i$, iff $\exists\ \Psi(S_i, S_j)$. We can see from this example that we need *both* the constraint variable C and Ψ, to indicate which constraints are passed between each stage.

We could have chosen *not* to represent the messages passed back upstream as constraints, because they tend to behave differently: constraints passed downstream typically *restrict* a search space, a feedback message passed upstream might result in an *enlarged* search space. A "feedback request" is usually made on failure to find a solution given the current constraints. The response to such a request may either be to output different constraints, or fewer constraints. Either of these responses will (as intended) change the search space, and may in fact increase the size of the space as compared to the space defined under the original set of constraints. Though it is unusual to increase the search space as messages are passed through the pipeline, the increased space may lead to the downstream stage finding a solution whereas without the feedback mechanism no solution would have been found. Despite the obvious utility of being able to backoff to a different set of constraints on failure in the downstream stage, feedback pipelines are not often implemented.

An example of a feedback mechanism in our vacation pipeline (Figure 1.1, p. 2) might be between the third and fourth stages: in the case that the hotel-search comes up empty, or returns very few options, then you might choose to go back to the city-search stage of the pipeline and expand the city-constraints to include the surrounding towns and areas. Some conceptual examples of feedback in NLP pipelines are: using a chunker to delimit the bottom brackets of a context-free parsing solution, then going back to the chunker if the next-stage parser failed to find a complete parse given the chunker's output constraints; similarly, using a supertagger to constrain a CCG parser, and allowing the CCG parser to go back and request more or different supertag constraints [59]; using a lexical translator to anchor phrases for phrasal translation, then returning to the lexicon for more translation options if the phrasal translator fails.

2.3.3 Iteration

In an *iteration* pipeline, the data makes multiple passes through the pipeline. Figure 2.5 shows a simple iteration pipeline, with one iteration passageway from the final stage of the pipeline back to the first stage. This pipeline could be defined as follows: $\mathcal{P}(\{S_1, S_2, S_3\}, \{O_1, O_2, O_3\}, \{I_1, I_2, I_3\}, \{C_1, C_2, C_3\}, \{\Psi(S_1, S_2), \Psi(S_2, S_3), \Psi(S_3, S_1)\},)$, with three stages, message-passing from stage 1 to 2, 2 to 3, and from stage 3 back to stage 1, and constraint sets output from stages 1, 2, and 3. Note that the iteration passageway does not necessarily have to be placed at the end of the pipeline, nor does it have to return to the beginning of the pipeline. However, an iterated message must pass from a downstream stage to an upstream stage lest the passage qualify as a feed-forward passage rather than iteration; there should be at least one stage between the two stages at the start

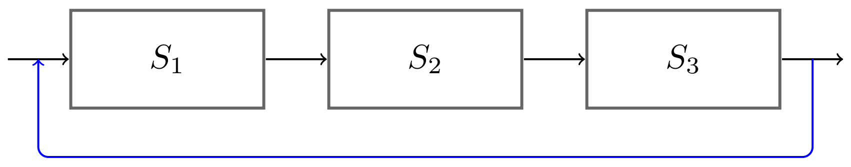


Figure 2.5: Iteration pipeline.

and end of the iteration passageway or the passage would be classified as a feedback mechanism rather than iteration. Thus, for an iteration passageway $\Psi(S_j, S_i)$, $j > i$ and $i + 1 \neq j$.

An example of iterating through our vacation pipeline (Figure 1.1, p. 2) might be that after seeing the hotel results returned by the final stage, you decide that only a five-star hotel will do for your vacation, and so you return to the beginning of the search with this new constraint. Ruling out all but five-star hotels will probably rule out some parts of the world as vacation locations, and so your second pass through the pipeline will be constrained differently than the first, and may change your decision process, although you might also end up with the same solution as you did after the first pass.

Iteration pipelines in NLP are rare, though Chapter 4 will show that iteration can be a surprisingly successful technique for improving pipeline performance. Two examples of NLP iteration pipelines are: using information from a full parse output by a reranking stage at the end of a parsing pipeline to constrain a first-stage chunker or POS-tagger in a second pass through the entire pipeline; and using the translation sequences output by a reranking stage at the end of a translation pipeline to constrain, in a second pass through the decoding process, the phrases extracted at an early stage of the pipeline.

2.3.4 Message Source

The source of the messages passed as constraints in the pipeline can affect pipeline performance. This section will discuss single- and multi-source pipelines, sampling techniques, and constraint generation and manipulation.

Single-Source vs Multi-Source Pipelines

In a *single-source* pipeline, data flows from a single upstream stage; thus there is a one-to-one relationship between all of the stages in the pipeline. In contrast, in a *multi-source* pipeline, the data flows from multiple upstream stages, which allows for many-to-one relationships between stages.

Figure 2.6 shows two simple single-source and multi-source pipelines. The single-source pipeline would be defined as: $\mathcal{P}(\{S_1, S_2, S_3\}, \{O_1, O_2, O_3\}, \{I_1, I_2, I_3\}, \{C_1, C_2\}, \{\Psi(S_1, S_3), \Psi(S_2, S_3), \Psi(S_3, S_4)\})$; note that $\forall j \in \Psi(S_i, S_j)$ there exists only one i. The multi-source pipeline would be defined as: $\mathcal{P}(\{S_1, S_2, S_3, S_4\}, \{O_1, O_2, O_3, O_4\}, \{I_1, I_2, I_3, I_4\}, \{C_1, C_2, C_3\}, \{\Psi(S_1, S_3), \Psi(S_2, S_3), \Psi(S_3, S_4)\})$; noting that here, for S_3, there are two constraining stages: S_1 and S_2.

The benefit of using multiple sources in a pipeline is that each source stage may constrain a different part of the search space, perhaps by providing partial solutions for different parts of the problem space. The multiple source stages may also, however, operate over the same part of the space. Thus there are several methods for combining constraints from multiple sources, including set union and intersection, and voting or distribution combination.

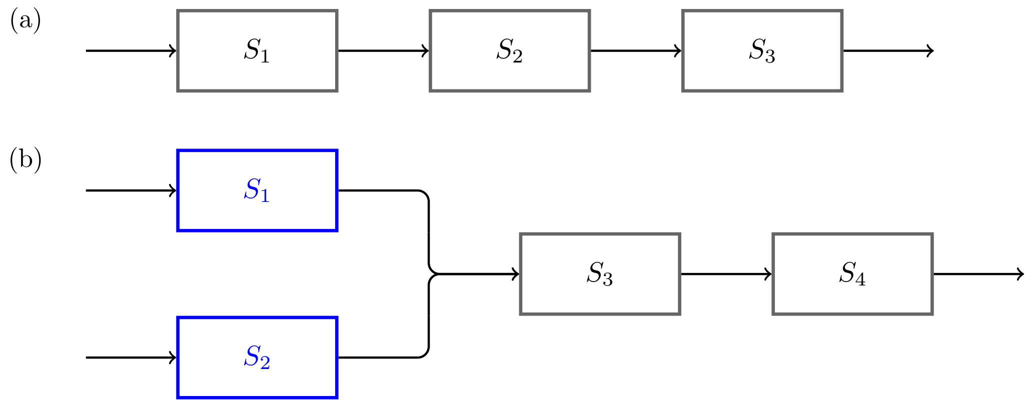


Figure 2.6: Compare (a) a single-source pipeline to (b) a multi-source pipeline.

Sampling

Sampling, a technique that will be discussed only briefly in this dissertation, is an interesting hybrid of extracting the final output solution by sampling the probability distribution of every stage in the pipeline. Finkel et al. [83] used Bayes-net sampling over the entire pipeline rather than using a single partial-solution candidate output from each stage to define the search space of the next stage. They did not compare to an n-best pipeline, where the output from each stage may be multiple possible solutions. This dissertation argues that n-best pipelines which include a probability distribution as part of each stage's output, perform similarly to a sampling pipeline but with a simpler implementation (see Section 2.2).

Message Manipulation

The messages passed from one stage to another in a pipeline can be systematically manipulated by an external process. There are several possible reasons to manipulate this data. One is to test the "ceiling" performance of the downstream stage, to determine how well downstream stages could perform if only the upstream stage had: included the correct solution within the space of possible solutions (see Section 2.5.2 for further techniques to address the problem of search errors), included a more diverse set of candidates, or defined a more regular set of candidates. These methods of manipulation and others will be discussed and tested thoroughly in Part III.

Another reason to manipulate data is to synthesize negative examples to be included in the training dataset. Some discriminative modeling approaches require negative examples in the training data. There are several different methods to synthesize negative examples for these models, some of which can significantly affect pipeline performance (as will be shown in Chapters 5 and 7).

2.4 Stage-Internal Attributes

The use of separate stages in pipeline systems is what makes pipelines such modular systems. Each stage acts as a "black-box" system in that stage-internal details, such as its search algorithm, feature space, objective function, etc., are hidden from other stages in the pipeline. The only interaction between stages is at the input and output of each stage. Thus, so long as the input/output of a stage conforms to the expected format, different systems can be plugged into the pipeline without re-designing the entire pipeline system. This plug-and-play modularity means that pipelines are flexible and re-usable systems.

A pipeline $\mathcal{P}$ may consist of any finite number of stages greater than one. Each of those stages has a number of stage-internal components, such as search method and objective. Each of these components, and their effect on pipeline design, will be discussed in the following sections.

2.4.1 Search Method

Search method is used here to encapsulate how a model searches through the space of possible solutions, and is closely related to the representation of the space, discussed in Section 2.2. The five main methods of search used in pipeline systems can be categorized as: *creation, extension, combination, selection*, and *ranking*.

The *creation* method is most easily compared to searching in an unconstrained space. The solution is created from scratch, built up over the raw input data. In this methodology, constraints are used to reject or score the solutions proposed (or created) by the model. Creation models are generally found at the beginning stages of a pipeline, typically built as stand-alone models rather than as part of a pipeline.

The *extension* method is similar to the creation method in that the solution is built up over the input data, but here the input is a partial solution (Section 2.2.1) rather than raw data. Thus the input partial solutions are extended to create the final output solution. Extension models are, by definition, never the first stage of a pipeline.

The *combination* method takes as input partial solutions, then combines those partial solutions in order to produce a complete solution. It can also take as input a set of complete solutions (Section 2.2.2), which are then typically broken into partial solutions and re-combined to form the complete solution for output. Unlike the extension method, where the input partial solutions under-specify the search space, here the input provides a full specification of the search space – each possible output solution could, given enough time and space, be enumerated from the input.

Both the *selection* method and the *ranking* method take sets of complete solutions as input. The selection method selects one of the inputs as its favored output, whereas the ranking method ranks and outputs the entire input set. Note that most of the popular so-called "reranking" systems (discussed more thoroughly in Chapter 3, p. 51) are actually used as selection systems: while the systems do indeed rank the entire input candidate set, only the highest-ranked candidate is considered to be the system's output. While it is usually straightforward to alter the system such that it outputs all of the the ranked candidates rather than just the favored one, one should also take into account the search objective used in training the system (see Section 2.4.2). For this dissertation, we define selection stages as being trained with a one-best search objective, such that the model is rewarded for finding a more-correct candidate; ranking stages are trained with a ranking objective, such that the model is rewarded for finding a ranking that is closer to the true ranking of the input candidates.

Different search methods can be affected in different ways by the characteristics of a search space. For example, the diversity of candidates has been mentioned in literature regarding combination (or "voted recombination") models [189, 191], but rarely (though *c.f.* [152]) in regards to selection models. Different search methods may be robust to changes in search space coverage, for example, but sensitive to changes in the quality characteristics (such as one-best and oracle-best accuracy rates). Or a search method may be more sensitive to changes in the quality of the space if the coverage of the space is more sparse. Current practices simply ignore these possible interactions, possibly to the detriment of pipeline performance.

Training and Testing

In designing a pipeline system, the choice of different search methods can also have an effect on experimental design and execution. Some methods will need to be trained only once in order to be tested under different conditions, whereas others will need to be re-trained for each testing condition. Methods that require a fully-specified space on input (i.e., combination, selection, and ranking) are often trained as discriminative models, discriminating between the positive and negative examples. If different examples are provided, say by altering an upstream stage to produce different output thus changing the input to the current stage, then the model should be re-trained. On the other hand, any method which assumes an unconstrained search space on input would not need to be re-trained even if output from an upstream stage changes.

One aspect of pipelines that is often-overlooked is that the pipeline may differ between train-time and test-time. Stages may be added to or removed from the pipeline between training and testing. Stage-internal components may also be changed. For example, a stage trained as a creation model may then be forced to take constraints on its input at test-time, thus altering the stage to become a extension, selection, or ranking model in testing. Note that while it is possible to change methods between training and testing, in practice it is rarely done (Chapter 3) although this dissertation will show (Chapter 5) that altering the search method between training and testing can have an interesting effect on pipeline performance.

One could actually take the concept of altering the pipeline between training and testing to the extreme, and design a pipeline where the stages were only constrained during training and unconstrained at test-time, or vice versa. Chapter 3 discusses some existing testing-only constrained pipelines, though pipelines that are constrained during both training-and-testing are much more common. Training-only constrained pipelines are extremely uncommon and may in fact be impossible in some cases: by virtue of limiting the search space to be explored during training, is it even possible for a model to truly explore an unconstrained space at test-time? This subject will be discussed in further detail in Chapter 5.

2.4.2 Search Objective

Search objective is used here to refer to the "target" solution in the space, or the solution that the stage-internal model is trained to find. The search objective is closely tied to the search method discussed in the previous section and the evaluation metrics discussed in Section 2.5. With combination and selection stages, the search objective is typically to place the highest model-weight on the true solution, or, if the true solution is not included as a candidate in the space, then on the *oracle* candidate (defined as the candidate closest to the true solution according to some evaluation metric). The objective of such a search may also be to weight candidates such that there is a large linear separation between the oracle candidate and the next highest-scoring candidate. With a ranking stage, the objective is to place a higher model-weight on higher-ranking candidates (according to the true ranking), and ideally to correctly rank all of the candidates in the space. The most important thing to note about the search objective is that it is typically defined stage-internally, and may not necessarily correlate to downstream objectives.

2.4.3 Thresholds and Beams

Previous sections have made the assumption that constraints placed on a system are stage-external. However, there are also stage-internal constraints. Two such examples are thresholding and beam search, which are popular techniques but not necessarily part of a pipeline architecture. The goals of thresholding and beam search are to improve the efficiency of a system or to reduce computational complexity, both of which are also goals of the pipeline architecture. However, these techniques can be implemented within a single-stage system, while pipeline systems were defined to have two or more stages (Section 2.1). External constraints, on the other hand, require two or more stages, and are the root of the pipeline architecture. Thus this dissertation will focus on understanding and improving external constraints that define the search space rather than stage-internal techniques to reduce search costs.

2.4.4 Complexity

The *complexity* of solutions output by stages in a pipeline typically increases. For example, a linear POS-tagger will output a tag sequence to constrain a cubic context-free parser. It is not, however, a necessary condition of a pipeline that sequential stages use models of monotonically increasing complexity. In fact, "post-processing" is very common in pipelines, where stages are appended to the end of a pipeline to clean up or otherwise filter the final output; such stages are typically very low complexity. By paying attention to the order of complexity within a pipeline, we might find an interesting pattern of performance as related to increasing, decreasing, or varying the order of complexity in a pipeline.

2.5 Evaluation

2.5.1 Intrinsic vs Extrinsic Evaluations

Intrinsic evaluations are a direct measure of the quality of each stage's output O_i, and allow for quick turnaround time in improving the accuracy of the current stage. However, improvements at a given stage may not translate to pipeline improvement. *Extrinsic* evaluations are a measure of the accuracy of the output at the end of the pipeline, and thus a measure of overall pipeline performance. An intrinsic evaluation of a POS-tagger would analyze the accuracy of the output POS-tag sequence as a stand-alone evaluation, whereas an extrinsic evaluation would analyze the effects of the tagger's output on the quality of a context-free parse constrained to be consistent with the POS-tag sequence. Figure 2.7 demonstrates the intrinsic and extrinsic evaluation points in a pipeline system.

While extrinsic measures would ideally be used to improve each stage in the pipeline, in practice it can be difficult to correlate changes in each individual stage to changes in the pipeline-final output. Furthermore, such extrinsic evaluations can be slow and costly, since a change at any stage must be propagated through the entire pipeline to determine its effects. Thus, intrinsic evaluations typically serve as an approximation, with varying degrees of effectiveness. One of the issues with using intrinsic evaluations is the the stage-internal objective may differ greatly from an objective that would better benefit the pipeline as a whole.

The availability of intrinsic evaluation affects the stage-internal search objective (Section 2.4.2); without an intrinsic evaluation metric, the search algorithm cannot optimize for the "truth" or

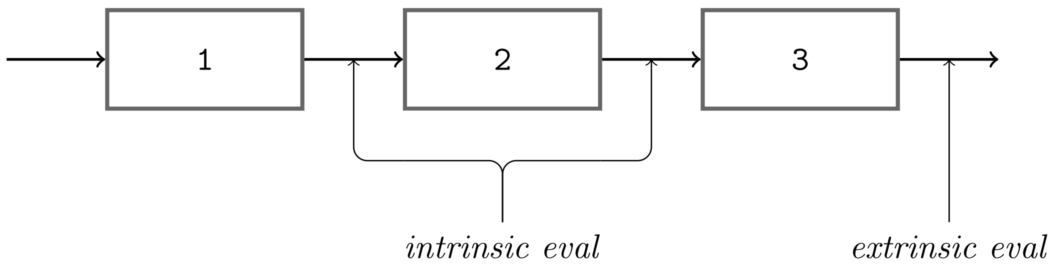


Figure 2.7: Intrinsic and extrinsic evaluation points.

the "oracle" candidates and must instead optimize for another objective. Most parsing pipelines include an intrinsic evaluation at each stage in the pipeline, whereas early stages of translation pipelines do not (see Chapter 3 for examples and further discussion). The use of intrinsic evaluations can be either beneficial or detrimental to pipeline performance, as will be shown in Chapter 6.

2.5.2 Error Types

This dissertation will aim to address two different type of errors in pipeline systems. The first we will term *search errors*, where there exists a higher-scoring candidate (according to the model), which has been excluded from the search space. This type of error is often called a "cascading error," which occurs when the output of one stage in the pipeline is erroneous and removes the correct solution from the search space and thus from consideration. The notion of cascading errors is often used in arguments against the utility of the pipeline architecture. However, there is another type of error as well, which we will term *model errors*, where the highest-scoring candidate (according to the model) is not the best candidate. NLP models are imperfect, resulting in model errors. Interestingly, the same framework that causes search errors can also resolve (or work around) model errors, as we will demonstrate in Chapter 4.

Typical pipeline systems do not include mechanisms for later stages in the pipeline to recover from mistakes made earlier in the pipeline (although recall our discussion about feedback pipelines in Section 2.3.2). Interestingly, the origin of several different methods for improving pipeline performance, outlined earlier in this chapter, can be traced to methods for addressing this problem of search errors. This section will briefly discuss these methods, highlighting the relevance to preceding sections.

The most common method is to increase the size of the search space. Increasing the size of a fully-specified space (Section 2.2.2) can be achieved by simply increasing the number of candidate solutions in the enumerated solution set. An under-specified space (Section 2.2.1) can be enlarged by imposing fewer constraints on it; either removing elements from the constraint set, or allowing disjunction in the set of constraints such that an acceptable solution must satisfy one, but not necessarily all, of the constraints in the set [121]. As an example, take the set of POS sequences in Figure 2.1 (p. 13). This set might be used to represent a complete-solution search space, in which case adding another POS sequence to the set would be a straightforward way to increase the size of the search space. These sequences could also be a disjoint set of constraints on the space of parse solutions, such that an acceptable parse would be one that contained the POS tags represented by any of the ten sequences in the constraint set. Although the problem of search errors can be alleviated by increasing the size of the search space, a larger search space can decrease the efficiency of a pipeline as well as causing other problems in the pipeline, as will be discussed in Section 6.1.2.

A second method is to move from using hard constraints to using soft constraints (Section 2.2.3), in which case no additional solution will be removed from consideration and thus there will be no search errors, though often at the cost of efficiency. To continue the example from above, an enumerated set of POS sequences could be used to indicate a preference for parses containing such sequences without explicitly disallowing parses containing different POS sequences.

The third method is to move away from the pipeline architecture entirely, and create a joint model by combining two or more pipeline stages into one all-encompassing model. Such a move will obviously remove the problem of search errors caused by upstream constraints but, again, at the cost of decreased efficiency. This third method is mentioned in order to acknowledge the fact that, for some applications, the accuracy benefit of a joint-conditional model outweighs the cost in efficiency, but will not be addressed in further detail in this dissertation since such systems are not pipelines and are therefore outside the scope of this research.

Identifying the presence of search errors is typically based on a hybrid of intrinsic and extrinsic evaluation. Search errors are identified based on mid-pipeline intrinsic evaluations, regardless of whether correcting the error would result in downstream (extrinsic) improvements. We will demonstrate in Chapter 4 (p. 73) another way to identify such search errors. Another point for consideration is that recent research has indicated that optimizing for a downstream oracle is a more attainable objective, thus lessening the need to correct for search errors. Chapter 6 will examine the effectiveness of alternate objective functions such as this for downstream performance.

2.6 Summary

In this chapter we defined a formal framework of pipeline systems, with four main elements: message passing, constraint representation, stage-internal attributes, and pipeline evaluation. In Section 2.3 we discussed three methods of passing messages in a pipeline, namely feed-forward, feedback, and iteration. We also discussed different sources of the messages, including single-source versus multi-source, sampling across a pipeline, and external manipulation of the messages. Chapter 3 (p. 48) will classify many existing implementations of pipeline systems based on the message-passing methods and sources of the pipeline. Chapter 4 will examine in detail our own implementation of an iterated parsing pipeline, along with the effects of iteration on pipeline performance.

In discussing the different types of constraint representations in Section 2.2, we examined the difference between partial-solution constraints and complete-solution constraints. Example pipelines using partial-solution constraints and complete-solution constraints will be given in Chapter 3 (p. 48). We also compared hard and soft constraints, and determined that focusing on hard constraints in this dissertation would allow us to more clearly see the effects of constraints on pipeline performance. Chapters 6 and 7 will analyze different characteristics of a search space defined by hard constraints. Since hard constraints are typically combined with soft constraints, Chapter 8 will demonstrate some of the effects of layering soft-constraint preferences over hard constraint–defined search spaces.

In Section 2.4 we discussed a number of different stage-internal attributes, including the search method and objective at each stage of a pipeline. Chapter 3 (p. 48) will provide a number of examples of pipeline systems that include some of the five different types of stages in pipeline systems: creation, extension, combination, selection, and ranking stages. Chapter 5 will discuss and dissect a number of assumptions about models in pipeline stages, and Chapter 7 will examine a few different methods for generating constraints in order to determine the effects of constraint characteristics on pipeline performance.

Finally, we discussed pipeline evaluation in Section 2.5. We illustrated the difference between intrinsic and extrinsic evaluations, and defined two different types of errors of interest in pipeline systems: search errors and model errors. Chapter 4 (p. 73) will discuss ways in which to identify the two different error types, what causes the errors, and a few ways to recover from them.

These four elements of message-passing, constraint representation, stage types, and pipeline evaluation provide the structure of our formal framework. The next chapter will use this framework to classify a large set of existing pipelines from parsing to machine translation to automatic speech recognition to image segmentation. This large-scale classification will serve as a benchmark for future researchers to classify their pipeline systems, as well as to assist them in determining the best type of pipeline system for their research problem.

Chapter 3

Background and Preliminaries

This chapter will begin with two extensive sections on notation, in order to establish a common notation for discussing the models and algorithms referenced frequently throughout this dissertation. These sections will define finite-state machines, context-free languages, Markov and log-linear models, as well as defining algorithms such as the Forward-Backward, Viterbi, CYK, and Inside-Outside algorithms.

Following these sections, we will present background information for the application areas of tagging, parsing, language modeling, automatic speech recognition (ASR), machine translation (MT), and image classification. We will examine pipeline systems from all of these application areas, to compare and contrast the different classes of pipelines implemented in these different areas to discover (a) *generalizable* techniques that are successful cross-application, and (b) *novel* pipeline implementations from one application area that might prove beneficial for other areas. Thus the application overview will provide a foundation for discussing pipelines from each of these areas.

The second half of the chapter will analyze a wide range of pipelines systems from these different application areas, and will classify each system according to our pipeline framework. Comparing and classifying pipelines from different applications will allow us to validate and improve upon our pipeline framework as patterns emerge in constraint representation, search algorithms implemented at various stages in a pipeline, and the objective measures and evaluations performed on a pipeline system. We will also see that despite being implemented for widely varying application areas, each with different objectives, data, and methods, many pipelines in these areas share underlying commonalities.

The chapter will conclude with detailed system descriptions of the implemented pipelines used for empirical trials throughout this dissertation.

3.1 Model Notation

In this section and the following (Section 3.2), we concisely introduce formal notation to discuss the models and algorithms commonly used in pipeline systems and referenced frequently throughout the dissertation. Much of the notation is borrowed from Roark and Sproat [187], and readers are referred there for details beyond what will be covered in these sections.

3.1.1 Finite-State Machines

A *finite-state automaton* (FSA), is a model composed of a finite number of states, transitions between those states, and actions. Formally, a finite-state automaton is a quintuple $M = (Q, s, F, \Sigma, \delta)$

where Q is a finite set of states, s is a designated initial state, F is a designated set of final states, Σ is an alphabet of symbols, and δ is a transition relation from a state-and-symbol pair in $Q \times (\Sigma \cup \epsilon)$ at the origin of the transition, to the destination state in Q.

A string in the language of the automaton is *matched* against the automaton as follows: starting in the initial state s, consume a symbol of the input string and match it against a transition leaving the current state. If a match is found, move to the destination state of the transition, and match the next symbol on the input with a transition leaving that state. If a final state $f \in F$ can be reached with all symbols of the input consumed, then the string is in the language of the automaton; otherwise it is not.

Finite-State Transducers (FSTs)

Analogous to finite-state automata are *finite-state transducers* (FSTs): whereas automata consume symbols on input to indicate an accept or reject state, transducers consume symbols on input and also produce symbols on output. With a transducer, an input string matches against the input symbols on the arcs, and at the same time the transducer outputs the corresponding output symbols. Formally, a finite-state transducer[1] is a quintuple $M = (Q, s, F, \Sigma_i \times \Sigma_j, \delta)$ where Q is a finite set of states, s is a designated initial state, F is a designated set of final states, Σ_i is an alphabet of input symbols, Σ_j is an alphabet of output symbols, and δ is a transition relation from $Q \times (\Sigma_i \cup \epsilon \times \Sigma_j \cup \epsilon)$ to Q.

FSTs are used to determine if an input string is in the *domain* of the relation, and if it is, computes the corresponding string, or set of strings, that are in the *range* of the relation. *Composition* of FSTs, denoted $\circ$, is to be understood in the sense of function composition. If g and f are two regular relations and x a string, then $[f \circ g](x) = g(f(x))$. In other words, the output of the composition of g and f on a string x is the output that would be obtained by first applying f to x and then applying g to the output of that first operation. We discussed in Section 2.3.1 that a *feed-forward pipeline* is equivalent to the composition of each of its stages, where each stage in the pipeline corresponds to a function in the composition: $g(f(x))$, or $(g \circ f)(x)$. Thus the composition of finite-state transducers is equivalent to a pipeline system; later in this chapter we will discuss several applications in speech and language processing of composed FSTs.

Weighted Finite-State Automata (WFSAs) and Transducers (WFSTs)

Finite-state automata and transducers can be extended to include *weights* or *costs* on the arcs. Such machines are termed *weighted finite-state automata* (WFSA) and *weighted finite-state transducers* (WFST). Formally, a weighted finite-state automaton is an octuple $A = (Q, s, F, \Sigma, \delta, \lambda, \sigma, \rho)$, where $(Q, s, F, \Sigma, \delta)$ is a finite-state automaton; an initial output function λ: $s \to \mathbb{K}$ assigns a weight to entering the automaton; an output function σ: $\delta \to \mathbb{K}$ assigns a weight to transitions in the automaton; and a final output function ρ: $F \to \mathbb{K}$ assigns a weight to leaving the automaton. Every transition $d \in \delta$ will have a label $t[d] \in (\Sigma \cup \epsilon)$, an origin stage $b[d] \in Q$, and a destination state $e[d] \in Q$. A *path* through the automaton then consists of k transitions $d_1, \ldots, d_k \in \delta$, where $e[d_j] = b[d_{j+1}]$ for all j, i.e., the destination state of transition d_j is the origin state of transition d_{j+1}. We will discuss the notion of paths again in Chapter 7 (Section 7.2.3).

Weighted finite-state transducers (WFSTs) are an obvious extension of finite-state transducers (FSTs) and weighted finite-state automata (WFSAs), where each transition relation $d \in \delta$ is associated with a weight.

[1] For simplicity here we assume a *two-way* FST.

Semirings

In weighting automata and transducers, we must also provide an interpretation of those weights to specify both how weights are to be combined *along* a path as well as how weights are to be combined *between* paths. If we had two identical paths in an automaton (with the same symbols on the arc labels of the paths), and we wanted to collapse those paths into a single path, then in order to ensure proper normalization, the weights of the two paths will be added together. Different kinds of weights—e.g., logs, probabilities, scores and costs—will have different ways of combining the weights along the path and between paths. *Semirings* provide a useful way to summarize the different interpretations of weights.

A semiring is a triple $(\mathbb{K}, \oplus, \otimes)$, where $\mathbb{K}$ is a set and $\oplus$ and $\otimes$ are binary operations on $\mathbb{K}$ where: $(\mathbb{K}, \oplus)$ is an additive operator (with 0 as its neutral element); $(\mathbb{K}, \otimes)$ is a product operator (with 1 as its neutral element); the product $\otimes$ distributes with respect to the sum $\oplus$, i.e., $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$; and for all a in $\mathbb{K}$, $a \otimes 0 = 0 \otimes a = 0$. When one extends a path in a weighted automaton, the resulting cost is calculated using the semiring $\oplus$ operator; the cost of combining two paths via intersection is calculated by using the semiring $\otimes$ operator. Common semirings used in NLP are the 'real' semiring $(+, \times)$, and the 'tropical' semiring $(\min, +)$. The $(+, \times)$ semiring is appropriate for use with *probability scores*: the probability of a path is obtained by multiplying along the path, and the probability of a set of paths is obtained by summing the probabilities of those paths. The $(\min, +)$ semiring is appropriate for use with *negative log probability scores*: weights along a path are summed, and the minimum of a set of paths is calculated, which can be used to determine the best scoring path in a weighted automaton, since lower scores are better with negative logs.

3.1.2 Context-Free Languages

The finite-state models described in the previous sections are insufficient to model some of the syntactic dependencies in natural language. We can use *context-free grammars* to encode some of these more-complex dependencies in natural language if we are willing to incur a significant efficiency cost, moving from the $O(n)$ complexity of finite-state inference algorithms to $O(n^3)$ complexity.

A context-free grammar (CFG) is defined as a quadruple, $G = (V, T, P, S^\dagger)$, consisting of a set of non-terminal symbols V, a set of terminal symbols T, a start symbol $S^\dagger \in V$, and a set of rule productions P of the form $A \rightarrow \alpha$, where $A \in V$ and $\alpha \in (V \cup T)^*$. Commonly used non-terminal symbols include NP (noun phrase), VP (verb phrase), and S (sentence); the latter is often specified as the start symbol $S^\dagger$ for a CFG. Terminal symbols are typically words (*a, prices, rise, the, trading, ...*). The set of nonterminals which expand to terminals (i.e., $v \in V : (v \rightarrow w)$ where $w \in T$) are called *pre-terminals*, and typically consist of part-of-speech tags. An example rule production is S → NP VP, which encodes the rule that an S non-terminal can consist of an NP non-terminal followed by a VP non-terminal. A CFG G defines the set of terminal (word) strings that can be derived from the start symbol: $\{\alpha \mid \alpha \in T^* \text{ and } S^\dagger \overset{*}{\Rightarrow} \alpha\}$, where one step in a derivation $\Rightarrow$ is defined as a *rewrites* relation between symbols on the left-hand side of the derivation rule and symbols on the right-hand side of the derivation. Given a CFG, we can define such a relation between sequences of terminals and non-terminals:

$$\beta A \gamma \Rightarrow \beta \alpha \gamma \qquad \text{for any } \beta, \gamma \in (V \cup T)^* \text{ if } A \rightarrow \alpha \in P. \tag{3.1}$$

These rewrites can take place in sequence, with the right-hand side of one rewrite in turn rewriting to something else. We let $\gamma \overset{k}{\Rightarrow} \beta$ denote a sequence of k rewrite relations, with γ on the left-hand side of the first relation, and β on the right-hand side of the last. We use $\gamma \overset{*}{\Rightarrow} \beta$ to signify that $\gamma \overset{k}{\Rightarrow} \beta$ for some $k \geq 0$, and thus a sequence γ *derives* a sequence β if $\gamma \overset{*}{\Rightarrow} \beta$.

Probabilistic Context-Free Grammars (PCFGs)

A weighted CFG $G = (V, T, S^\dagger, P, \rho)$ is a CFG plus a mapping $\rho : P \rightarrow \mathbb{R}$ from rule productions to real valued weights. A *probabilistic context-free grammar* (PCFG) is a weighted CFG with a probability assigned to each rule production in P, and defines a distribution over all strings that can be produced by the grammar. Given the context-free nature of these grammars, the probability distribution over a grammar can be described as a set of conditional probability distributions, where the left-hand side of a rule production is the conditioning context. Thus the probability of each production in the grammar is

$$\mathrm{P}(\mathrm{A} \rightarrow \alpha) = \mathrm{P}(\alpha|\mathrm{A}) \tag{3.2}$$

where the weight of each production in the grammar is defined such that for any non-terminal $A \in V$:

$$\sum_{A \rightarrow \alpha \in P} \rho(A \rightarrow \alpha) \quad = \quad 1 \tag{3.3}$$

in order to ensure proper normalization.

Chomsky Normal Form (CNF)

Certain forms of CFGs have beneficial computational properties, and transforming a given CFG G to a *weakly equivalent* CFG G' in a particular form with those properties can be essential for certain uses of the grammar (as we will see in Section 3.2.4). Two grammars G and G' are weakly equivalent if the language defined by G is equivalent to the language defined by G'; the grammars strongly equivalent if, ignoring any differences in the non-terminal labels V, they derive the same parse trees for any given string.

One widely used form of CFGs is the *Chomsky normal form* (CNF), and every CFG G has a weakly equivalent grammar G' in Chomsky normal form. A CFG is in CNF if all productions are of the form $A \rightarrow B\ C$ or $A \rightarrow w$ for $A, B, C \in V$ and $w \in T$. In other words, a CFG is in CNF if the only productions that are not binary have a single terminal item on the right-hand side, and terminals are only on the right-hand side of unary rules. Some of the algorithms we discuss in Section 3.2 require grammars to be in Chomsky Normal Form.

3.1.3 Hidden Markov Models (HMMs)

A *hidden Markov model* (HMM) is a special-case WFST: a statistical model in which the system being modeled is assumed to be a Markov process with unobserved states. In such a model, while the state is not directly visible, output dependent on the state is visible. The states of an HMM emit the observations according to some probability distribution, thus the output sequence generated by an HMM gives some information about the (hidden) sequence of states. In an HMM, each state τ_i represents a random variable at time i that can adopt any of a number of values; the conditional probability distribution of τ_i, given the values of the hidden variable τ at all times, depends only on the value of the hidden variable τ_{i-1}; the values at time $t-2$ and before have no influence. This is called the *Markov property.* Similarly, the value of the observed variable w_i only depends on the value of the hidden variable τ_i.

Given the parameters of an HMM model and a particular output sequence $w_1 \ldots w_k$, we can calculate the probability of observing that sequence as follows (under a Markov-1 assumption):

$$\mathrm{P}(w_1 \ldots w_k) = \sum_{\tau_1 \ldots \tau_k \in \mathcal{T}^k} \left(\prod_{i=1}^{k} \mathrm{P}(\tau_i|\tau_{i-1})\ \mathrm{P}(w_i|\tau_i) \right) \tag{3.4}$$

where $\mathcal{T}$ is the set of possible output states; $\mathrm{P}(\tau_i|\tau_{i-1})$ is the transition probability from state τ_{i-1} to state τ_i; $\mathrm{P}(w_i|\tau_i)$ is the observation probability, or the probability of generating the observation w_i given state τ_i. We can also find the state sequence most likely to have generated the output sequence:

$$\begin{aligned}
\hat{\tau_1}\dots\hat{\tau_k} &= \operatorname*{argmax}_{\tau_1\dots\tau_k\in\mathcal{T}^k} \mathrm{P}(\tau_1\dots\tau_k|w_1\dots w_k) && (3.5)\\
&= \operatorname*{argmax}_{\tau_1\dots\tau_k\in\mathcal{T}^k} \frac{\mathrm{P}(w_1\dots w_k|\tau_1\dots\tau_k)\ \mathrm{P}(\tau_1\dots\tau_k)}{\mathrm{P}(w_1\dots w_k)} \quad \text{using Bayes rule} && (3.6)\\
&= \operatorname*{argmax}_{\tau_1\dots\tau_k\in\mathcal{T}^k} \prod_{i=1}^{k} \mathrm{P}(\tau_i|\tau_1\dots\tau_{i-1})\ \mathrm{P}(w_i|\tau_1\dots\tau_i, w_1\dots w_{i-1}) && (3.7)\\
&\approx \operatorname*{argmax}_{\tau_1\dots\tau_k\in\mathcal{T}^k} \prod_{i=1}^{k} \mathrm{P}(\tau_i|\tau_{i-1})\ \mathrm{P}(w_i|\tau_i). && (3.8)
\end{aligned}$$

We can also derive the maximum likelihood estimate of the parameters of the HMM given a dataset of output sequences. Performing these calculations in a brute force manner is computationally intractable because they require either summing over all possible state sequences or finding the maximum over all possible state sequences. However, these computations can be performed efficiently using dynamic programming, as discussed in Section 3.2.

3.1.4 Conditional Markov Models

Conditional Markov models provide an alternative to HMMs, where the HMM transition probability $\mathrm{P}(\tau_i|\tau_{i-1})$ and observation probability $\mathrm{P}(w_i|\tau_i)$ are replaced by a single probability function:

$$\mathrm{P}(\tau_i|\tau_{i-1}, w_i) \qquad (3.9)$$

of the current state τ_i given the previous state τ_{i-1} *and* the current observation w_i. In these models, the observations are given and thus only the probability of the state sequence they induce is calculated. Conditional Markov models move away from the *generative, joint probability* parameterization of HMMs to a *conditional* model that represents the probability of reaching a state given an observation and the previous state. These conditional probabilities are typically specified by exponential models based on arbitrary observation features.

An exponential model for a sequence of words and tags $\mathrm{w}\boldsymbol{\tau} = w_1\tau_1\dots w_k\tau_k$ has the form:

$$\mathrm{P}(\mathrm{w}\boldsymbol{\tau}) = \frac{\exp(\Phi(\mathrm{w}\boldsymbol{\tau})\cdot\bar{\alpha})}{Z(\Sigma^*)} = \frac{\exp(\sum_{i=1}^{n}\phi_i(\mathrm{w}\boldsymbol{\tau})\alpha_i)}{Z(\Sigma^*)} \qquad (3.10)$$

where ϕ_i is an n-dimensional vector encoding the features derived from the sequence $\mathrm{w}\boldsymbol{\tau}$; each feature has a corresponding parameter weight α_i. The denominator Z simply ensures normalization:

$$Z(\Sigma^*) = \sum_{\mathrm{w}\in\Sigma^*}\sum_{\boldsymbol{\tau}\in\mathcal{T}^{|\mathrm{w}|}} \exp\left(\sum_{i=1}^{n}\phi_i(\mathrm{w}\boldsymbol{\tau})\alpha_i\right) \qquad (3.11)$$

where $\mathcal{T}^{|\mathrm{w}|}$ is the set of tag sequences of length $|\mathrm{w}|$.

If instead of a joint model we use a conditional model, only the normalization changes and the equations become:

$$\mathrm{P}(\boldsymbol{\tau}\mid\mathrm{w}) = \frac{\exp(\sum_{i=1}^{n}\phi_i(\mathrm{w}\boldsymbol{\tau})\alpha_i)}{Z(\mathrm{w})} \qquad (3.12)$$

with the normalization constant

$$Z(\mathrm{w}) \quad = \quad \sum_{\boldsymbol{T}\in\mathcal{T}^{|\mathrm{W}|}} \exp(\sum_{i=1}^{n} \phi_i(\mathrm{w}\boldsymbol{T})\alpha_i) \tag{3.13}$$

Taking the log of these probabilities gives:

$$\log \mathrm{P}(\boldsymbol{T} \mid \mathrm{w}) \quad = \quad \sum_{i=1}^{n} \phi_i(\mathrm{w}\boldsymbol{T})\alpha_i - \log Z(\mathrm{w}) \tag{3.14}$$

which is essentially the linear combination of weighted features, discussed in the next section.

3.1.5 Log-Linear Models

Log-linear models associate arbitrary observation features with parameter weights. The benefit of these models over HMMs is that arbitrary, mutually dependent features can be used in the model, and are not limited by the conditional independence assumptions in the HMM model structure, but rather by the computational complexity of the log-linear model's structure. Log-linear models have been applied to POS-tagging [173, 65], shallow parsing [137], context-free parsing (and beyond) [174, 58], language modeling [9, 186, 192], and a range of reranking tasks [65, 44, 116, 199].

In a log-linear model, real-valued observation features are extracted from a solution candidate and stored in an n-dimensional vector of feature values Φ. The candidate is assigned a score by taking the dot product of Φ with the corresponding feature weight vector $\bar{\alpha}$. For finite-state models, the features typically remain quite similar to the HMM features, including tag-tag and tag-word pairs. However, more complicated feature types can be used, for example, the previous tag and word. Log-linear models are typically described as including features of a particular *type* or *template*, such tag-word pairs, but the templates are instantiated with very specific patterns such as: "the word *prices* with tag NNS followed by the word *rose* tagged with VBD," or "the word *light* tagged with JJ followed by an unknown word ending in *-ing* tagged with VBG."

There are various parameter estimation methods to estimate the weights α_i for some optimization objective; Gao et al. [93] present a comprehensive study of parameter estimation methods. Two such methods are defined in the next two sections.

Perceptron Algorithm

We follow Collins [65] in defining a *perceptron algorithm* for Markov models. Intuitively, the perceptron algorithm iteratively selects the best-scoring candidate (according to the current model parameterizations), then decreases that candidate's scores by penalizing its features and rewarding the features of the true reference candidate. The algorithm assumes: supervised training examples (x, y) where x is the input and y is the true output; a function GEN which enumerates a set of solution candidates for an input x; a representation function Φ which maps each $(x, y) \in \mathcal{X} \times \mathcal{Y}$ to a feature vector $\Phi(x, y) \in \mathbb{R}^d$; and a parameter vector $\bar{\alpha} \in \mathbb{R}^d$ to store the learned weights of each feature.

Since only the best-scoring candidate z is evaluated for the perceptron algorithm, we can rewrite Equation 3.14 without the normalization and call it a weight (W) instead of a log probability:

$$\mathrm{W}_t(\boldsymbol{T} \mid \mathrm{w}) \quad = \quad \sum_{i=1}^{n} \phi_i(\mathrm{w}\boldsymbol{T})\alpha_i^t \tag{3.15}$$

where α_i^t denotes the feature-weight vector α_i after t updates. The perceptron algorithm initializes all α_i^0 as 0. For each input (x, y) in the training set, the perceptron algorithm generates a possible solution z=GEN(x) for the input x, then penalizes the features of the generated solution z and rewards the features of the true solution y by incrementing or decrementing (as appropriate) the parameter values in $\bar{\alpha}$ that correspond to the z and y features. Thus after every training example, the parameters α_i^t are updated:

$$\alpha_i^t \quad = \quad \alpha_i^{t-1} + \phi_i(y) - \phi_i(z). \tag{3.16}$$

The basic algorithm updates after each sentence for some number of iterations through the training corpus. Collins [65] presents more details on this approach, as well as empirical evidence on POS tagging and NP-chunking that the perceptron improved over results using a MaxEnt tagger. Our CSLUt tagger [109, 110], used extensively throughout this dissertation and discussed in detail in Section 3.10.1, is a perceptron model.

MaxEnt Models

Maximum Entropy (MaxEnt) models are log-linear models estimated using general numerical optimization techniques to match the expected frequency of features with the actual frequency of features. The expected frequencies of features are estimated using techniques similar to the Forward-Backward algorithm which will be presented in Section 3.2.3. In a simple MaxEnt model, each tagging decision is modeled as a separate classification problem:

$$\log \mathrm{P}(\boldsymbol{\tau} \mid \mathrm{w}) \quad = \quad \sum_{j=1}^{k} \log \mathrm{P}(\tau_j \mid \mathrm{w} t_1 \ldots t_{j-1}). \tag{3.17}$$

Under a Markov order-1 assumption, the classification calculation becomes:

$$\log \mathrm{P}(\tau_j \mid \mathrm{w} t_1 \ldots t_{j-1}) \quad = \quad \sum_{i=1}^{n} \phi_i(\mathrm{w}\tau_{j-1}\tau_j)\alpha_i - \log Z(\mathrm{w}, \tau_{j-1}). \tag{3.18}$$

Ratnaparkhi [173] presented a MaxEnt POS-tagger, which is still a widely used, competitive tagger, and the state-of-the-art Charniak [41] parser is a "MaxEnt-inspired" model. Furthermore, the reranking stage of the Charniak and Johnson [44] parsing pipeline, used extensively throughout this dissertation and described in detail in Section 3.10.2, is based on a MaxEnt model.

Overfitting

One problem with log-linear models is that the models, if allowed to train until they converge, will "memorize" the training data and be unable to generalize to new data; this is called *overfitting*. For the perceptron algorithm, techniques known as voting or averaging are used to avoid overfitting. Collins [65] showed that using an *averaged* perceptron provided improved results; any references to a perceptron model in the remainder of this dissertation may be assumed to mean an averaged perceptron model. For the MaxEnt technique, the most common technique to avoid overfitting is called *regularization*, which associates a penalty with moving the parameter weights away from zero.

3.2 Algorithmic Notation

In this section we introduce notation for a number of dynamic programming algorithms used extensively throughout this dissertation and in NLP. *Dynamic programming* is a general method for finding globally optimal solutions by solving a sequence of subproblems, and in particular here we are referring to *top-down dynamic programming*, in which the results of certain calculations are stored and re-used later because the same calculation is a sub-problem in a larger calculation. In scenarios where one is searching from among a very large (exponential) set of solutions, dynamic programming can make the search tractable by taking much less time than naïve search methods. We will see dynamic programming techniques in many NLP applications, including Viterbi decoding and chart parsing.

3.2.1 Forward Algorithm

The *Forward algorithm* is used to efficiently calculate the sum over many distinct state sequences, taking advantage of the Markov property of HMMs to share sub-calculations needed for the summation. For a word sequence $w_1 \dots w_k$, a tagset $\mathcal{T} = \{\tau_i : 1 \leq i \leq m\}$, and a vocabulary $\Sigma = \{v_i : 1 \leq i \leq n\}$, let us define C_t as the random variable of the class associated with word w_t for a time index t in the sequence $w_1 \dots w_k$. The random variable C_t can take as its value any of the tags from $\mathcal{T}$. Let a_{ij} be defined as follows

$$a_{ij} \quad = \quad \mathrm{P}(C_t = \tau_j \mid C_{t-1} = \tau_i) \qquad \text{for} \;\; \tau_i, \tau_j \in \mathcal{T}, 1 < t < k \tag{3.19}$$

i.e., the probability that the class for word w_t is τ_j given that the class for word w_{t-1} is τ_i. For example, if τ_i = JJ (adjective) and τ_j = NN (noun), then a_{ij} is the probability of the tag NN labeling a word immediately following a word labeled with tag JJ. We will denote by $a_{\cdot\cdot}$ the set of all a_{ij} transition probabilities for a particular tagset $\mathcal{T}$. For simplicity, let a_{0j} and a_{i0} be defined as follows:[2]

$$a_{0j} = \mathrm{P}(C_1 = \tau_j \mid C_0 = \text{<s>})$$
$$a_{i0} = \mathrm{P}(C_{k+1} = \text{</s>} \mid C_k = \tau_i)$$

where <s> and </s> are special symbols that only occur at the beginning and end of the sequences, respectively. Next, let $b_j(w_t)$ be defined as:

$$b_j(w_t) \quad = \quad \mathrm{P}(w_t \mid C_t = \tau_j) \qquad \text{for} \;\; w_t \in \Sigma, \tau_j \in \mathcal{T}, 1 \leq t \leq k \tag{3.20}$$

such that if w_t = "light" and τ_j = JJ, then $b_j(w_t)$ is the probability of the word "light" given that the word is in the POS-tag class JJ.[3] We will denote by $b_\cdot(\cdot)$ the set of all $b_j(w_t)$ transition probabilities for a particular tagset $\mathcal{T}$ and vocabulary Σ.

We can now define a recursive value $\alpha_j(t)$, known as the *forward probability*, which is the probability of seeing the initial observed sequence $w_1 \dots w_t$ with tag τ_j at time t. Let $\alpha_0(0) = 1$. Then for $t \geq 1$, $\alpha_0(t) = 0$ and for $j > 0$:

$$\alpha_j(t) \quad = \quad b_j(w_t) \left(\sum_{i=0}^{|\mathcal{T}|} \alpha_i(t-1) a_{ij} \right). \tag{3.21}$$

Figure 3.1 presents an efficient algorithm for calculating the probability of an input string, given an HMM POS-tagging model, by calculating the sum over all possible tag sequences using this definition of the forward probability. Again $a_{\cdot\cdot}$ and $b_\cdot(\cdot)$ in the input represent the given probability models.

The Forward algorithm can be applied regardless of the order of the Markov assumption placed on the HMM by encoding the appropriate amount of history in to the state space; for example,

[2] In the presentation of these algorithms, we do not assume that w_k is </s> in a sequence $w_1 \dots w_k$.

[3] Note that $b_j(w_t) = b_j(v_i)$ if $w_t = v_i$, so there are two possible indices that we can use for words in the vocabulary, but to avoid complicating notation, we will usually use the time index w_t.

```
Forward($w_1 \ldots w_k$, $\mathcal{T}$, $a_{\cdot\cdot}$, $b_{\cdot}(.)$)

1  $\alpha_0(0) \leftarrow 1$
2  for $t = 1$ to $k$ do
3      for $j = 1$ to $|\mathcal{T}|$ do
4          $\alpha_j(t) \leftarrow b_j(w_t)\left(\sum_{i=0}^{|\mathcal{T}|} \alpha_i(t-1)a_{ij}\right)$
5  return $\sum_{i=1}^{|\mathcal{T}|} \alpha_i(k)a_{i0}$
```

Figure 3.1: Pseudocode of the Forward algorithm for an HMM POS tagger.

creating composite tags to encode the class of the current word as well as the previous $n-1$ classes, where n is the Markov order. Then, instead of τ_j, we would have $\tau_{j_1 \ldots j_n}$, where τ_{j_n} is the tag of the current word in the original tagset. Then the Forward algorithm, as well as the Viterbi and the Forward-Backward algorithms that will be presented in the next two sections, can be used for Markov models of arbitrary order.

3.2.2 Viterbi Algorithm

The *Viterbi algorithm* is a dynamic programming algorithm originally used to efficiently determine the most likely sequence of hidden states for a sequence of observed events given an HMM model; it has also been used with other models. The algorithm assumes that both the observed events and hidden states are in a *sequence*, that these two sequences are aligned such that an instance of an observed event corresponds to exactly one instance of a hidden state, and computing the most likely hidden sequence up to the point t must depend only on the observed event at point t and the most likely sequence at point $t-1$ (the Markov assumption).

The Viterbi algorithm is very similar to the Forward algorithm discussed in the previous section. The definitions of a_{ij} and $b_j(w_t)$ are identical, but $\alpha_j(t)$ is re-defined as follows. Let $\alpha_0(0) = 1$. For $t \geq 1$, $\alpha_0(t) = 0$ and for $j > 0$:

$$\alpha_j(t) \quad = \quad b_j(w_t)\left(\max_{i=0}^{|\mathcal{T}|}\left(\alpha_i(t-1)a_{ij}\right)\right) \tag{3.22}$$

The only change from Equation 3.21 is to replace the summation with a max, i.e., instead of summing across all possible previous classes, we take just the highest score. The second difference from the Forward algorithm is that in addition to calculating this recursive score, we need to keep track of where the max came from, in order to re-construct the maximum likelihood path at the end. This is called a *backpointer*, since it points back from each state in the sequence to the state at the previous time-step that provided the maximum score. We define $\zeta_j(t)$ to denote the backpointer for tag j at time t. Then the Viterbi algorithm is shown in Figure 3.2.

The perceptron models presented in Section 3.1.5 (p. 29) use a Viterbi-like algorithm; in contrast, the MaxEnt models (Section 3.1.5, p. 30) make use of an algorithm very similar to the Forward-Backward algorithm presented in the next section.

3.2.3 Forward-Backward Algorithm

The *Forward-Backward algorithm* uses the Forward algorithm defined in Section 3.2.1 to perform a forward pass over the input string, then performs a backward pass—in the other direction—to calculate the *backward* probability. The a_{ij} and $b_j(w_t)$ definitions for the Forward-Backward algorithm remain the same as for the Forward algorithm, as does the definition of the forward probability $\alpha_j(t)$ (which uses the sum, not the max of the Viterbi algorithm). Let us now define

```
VITERBI($w_1 \ldots w_k$, $\mathcal{T}$, $a_{\cdot\cdot}$, $b_{\cdot}(\cdot)$)

 1  $\alpha_0(0) \leftarrow 1$
 2  for $t = 1$ to $k$ do
 3      for $j = 1$ to $|\mathcal{T}|$ do
 4          $\zeta_j(t) \leftarrow \text{argmax}_i\,(\alpha_i(t-1)a_{ij})$
 5          $\alpha_j(t) \leftarrow \max_i\,(\alpha_i(t-1)a_{ij})\,b_j(w_t)$
 6  $\zeta_0(k+1) \leftarrow \text{argmax}_i(\alpha_i(k)a_{i0})$
 7  $\rho(k+1) \leftarrow 0$
 8  for $t = k$ to 1 do                          ▷ traceback with backpointers
 9      $\rho(t) \leftarrow \zeta_{\rho(t+1)}(t+1)$
10      $\hat{C}_t \leftarrow \tau_{\rho(t)}$
11  return $\hat{C}_1 \ldots \hat{C}_k$
```

Figure 3.2: Pseudocode of the Viterbi algorithm for HMM decoding.

another recursive value $\beta_j(t)$ as follows. Given a word string $w_1 \ldots w_k$, let $\beta_i(k) = a_{i0}$. For $0 \leq t < k$:

$$\beta_i(t) = \sum_{j=1}^{|\mathcal{T}|} \beta_j(t+1)a_{ij}b_j(w_{t+1}). \tag{3.23}$$

The forward probability $\alpha_i(t)$ is the probability of seeing the initial sequence $w_1 \ldots w_t$ with tag τ_i at time t, whereas the backward probability $\beta_i(t)$ is the probability of seeing the remaining sequence $w_{t+1} \ldots w_k$ *given* tag τ_i at time t. Note that because $\beta_i(t)$ depends on $\beta_i(t+1)$, therefore calculation of the backward probability is performed from right-to-left (as compared to the calculation of the forward probability which is performed from left-to-right).

The product of the forward and backward probabilities, $\alpha_i(t)\beta_i(t)$, is the probability of seeing the entire sequence $w_1 \ldots w_k$ with tag τ_i at time t. From this product we calculate $\gamma_i(t)$, the posterior probability of tag τ_i at time t given the entire string $w_1 \ldots w_k$:

$$\gamma_i(t) = \frac{\alpha_i(t)\beta_i(t)}{\sum_{j=1}^{|\mathcal{T}|} \alpha_j(t)\beta_j(t)}. \tag{3.24}$$

The value $\gamma_i(t)$ is equivalent to the sum of the probabilities of all tag sequences with tag τ_i at time t. In addition, we can calculate $\xi_{ij}(t)$, the probability of tag τ_i at time t and tag τ_j at time $t+1$ given the entire string $w_1 \ldots w_k$:

$$\xi_{ij}(t) = \frac{\alpha_i(t)a_{ij}b_j(w_{t+1})\beta_j(t+1)}{\sum_{l=1}^{|\mathcal{T}|}\sum_{m=1}^{|\mathcal{T}|} \alpha_l(t)a_{lm}b_m(w_{t+1})\beta_m(t+1)}. \tag{3.25}$$

Bilmes [21] provides a detailed derivation of these values.

The values of γ and ξ are used in the FORWARD-BACKWARD algorithm, shown in Figure 3.3, as an alternative approach for decoding an input sequence of observations. The first part of the FORWARD-BACKWARD algorithm is identical to the FORWARD algorithm presented in Figure 3.1. During the backward pass, once the backward probabilities have been calculated for all tags at time t, we can also calculate the $\gamma_{\cdot}(t)$ and $\xi_{\cdot\cdot}(t)$ values. (Note that a special case must be made for time k when calculating $\xi_{\cdot\cdot}(k)$.) Once the posterior probabilities $\gamma_{\cdot}(\cdot)$ have been calculated, we can then select the tag at each time step with the maximum γ value at that time. In some cases, the output state sequence returned by the Forward-Backward algorithm will differ from the

```
Forward-Backward($w_1 \dots w_k$, $\mathcal{T}$, $a_{\cdot\cdot}$, $b_{\cdot}(\cdot)$)

 1 $\alpha_0(0) \leftarrow 1$
 2 for $t = 1$ to $k$ do                                    ▷ Forward pass
 3     for $j = 1$ to $|\mathcal{T}|$ do
 4         $\alpha_j(t) \leftarrow b_j(w_t) \left(\sum_{i=0}^{|\mathcal{T}|} \alpha_i(t-1)a_{ij}\right)$
 5 for $t = k$ to 1 do                                     ▷ Backward pass
 6     for $i = 1$ to $|\mathcal{T}|$ do
 7         if $t < k$
 8             then $\beta_i(t) \leftarrow \sum_{j=0}^{|\mathcal{T}|} \beta_j(t+1)a_{ij}b_j(w_{t+1})$
 9         else $\beta_i(t) \leftarrow \alpha_{i0}$
10     for $i = 1$ to $|\mathcal{T}|$ do
11         $\gamma_i(t) \leftarrow \frac{\alpha_i(t)\beta_i(t)}{\sum_{j=0}^{|\mathcal{T}|}\alpha_j(t)\beta_j(t)}$
12         if $t < k$
13             then $\xi_{i0}(t) \leftarrow 0$
14                  for $j = 1$ to $|\mathcal{T}|$ do
15                      $\xi_{ij}(t) \leftarrow \frac{\alpha_i(t)a_{ij}b_j(w_{t+1})\beta_j(t+1)}{\sum_{k=1}^{|\mathcal{T}|}\sum_{l=1}^{|\mathcal{T}|}\alpha_k(t)a_{kl}b_l(w_{t+1})\beta_l(t+1)}$
16         else $\xi_{i0}(t) \leftarrow \gamma_i(t)$
17              for $j = 1$ to $|\mathcal{T}|$ do
18                  $\xi_{ij}(t) \leftarrow 0$
19 return $(\gamma_{\cdot}(\cdot), \xi_{\cdot\cdot}(\cdot))$
```

Figure 3.3: Pseudocode of the Forward-Backward algorithm for HMMs.

state sequence returned by the Viterbi algorithm, and relatively large improvements in accuracy can be obtained using such techniques instead of Viterbi decoding.

Note that these algorithms, which are presented here for use with HMMs (Section 3.1.3), can also be used with conditional Markov models (Section 3.1.4).

3.2.4 CYK Parsing Algorithm

The *Cocke-Younger-Kasami* (CYK) algorithm [61, 122, 217] is a dynamic programming algorithm used to find in $O(n^3)$ the maximum likelihood parse given a particular PCFG. As with other dynamic programming algorithms, CYK parsing relies upon being able to break down the optimal solution into independent optimal sub-solutions. Dynamic programming can be used to find the maximum likelihood parse given a PCFG because in a PCFG, the probability of the children (on the right-hand side of a rule production) is conditioned only on the category of the parent (on the left-hand side of a rule production). Thus, if the maximum likelihood parse for the string contains an NP node over the first three words of the sentence, that NP must be the maximum likelihood constituent of all possible ways of constructing an NP over those words.

CYK parsing operates over a two-dimensional table, commonly referred to as a *chart* [123]. Figure 3.4 shows the parse tree from Figure 1.2(c) (p. 4) as it would be represented in a chart. Each entry in the chart is a *labeled bracket*. Each parse constituent—or node in a parse tree—can be uniquely identified as a set of labeled brackets. Labeled brackets consist of three parts: the label, start-position, and span of the bracket. The start-position is based on word (terminal) indices, and the *span* of a node represents the number of words covered by the node. The S node from Figure 1.2 would be written as (S,1,6), with 'S' as its label, starting at the first word, with a span of 6. Labeled brackets are used extensively for CYK parsing, and for many other parsing

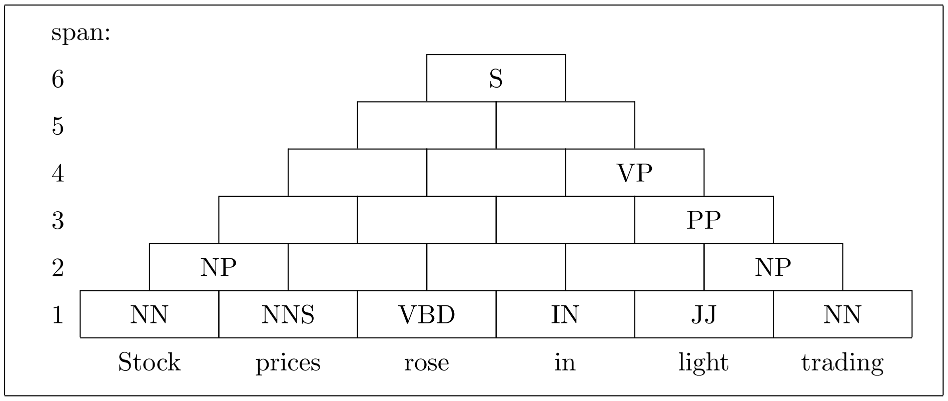


Figure 3.4: A parse represented in chart form, for CYK parsing.

algorithms including parse evaluation (discussed in Section 3.3.2, p. 41), to represent a parse tree.

To parse an input sentence given a PCFG in Chomsky normal form (CNF),[4] the CYK algorithm constructs a *chart*, such as that shown in Figure 3.4. The bottom layer of the chart represents nodes with a span of 1; higher layers represent nodes of greater spans. The cells of the chart represent unique start and end node locations; a cell may contain many nodes with the same span but with different labels. Cells in the chart are typically visited in a left-to-right, bottom-up order, guaranteeing that all smaller spans are visited prior to visiting larger spans. Each entry in a cell is a labeled bracket with "backpointers" to the node's children, which can be used recursively to build the entire subtree covered by the chart entry. Entries in the top cell represent parse trees that cover the entire sentence; the entry in the top cell with the label $S^\dagger$ and the highest probability represents the "best" (highest-probability) parse for the sentence.

To describe the CYK algorithm, we adopt notation similar to what was used for the Viterbi algorithm in Section 3.2.2. For non-terminals $A_i A_j A_k \in V$, let $a_{ijk} = \mathrm{P}(A_i \to A_j\ A_k)$, which is analogous to the transition probabilities in the Viterbi algorithm. Let $b_j(w_t) = \mathrm{P}(A_j \to w_t)$, analogous to the Viterbi observation probabilities. For each non-terminal in each cell we can store just a probability and a backpointer; we let $\alpha_i(x, s)$ denote the probability for non-terminal i with start index x and span s; and $\zeta_i(x, s)$ denote the backpointer to the children of the non-terminal.

The CYK algorithm is shown in Figure 3.5. Note that this algorithm is very similar to the Viterbi algorithm in Figure 3.2, except that the dynamic programming is performed over chart cells rather than word positions. Second, the argmax in line 9 of the algorithm involves a search over all possible mid-points m, which ranges from $x{+}1$ to $x{+}s{-}1$, for a total of $s{-}1$ points. Thus there are three nested loops (s, x, m) that depend on the length of the string n; therefore this algorithm has a worst case complexity of $O(n^3)$.

In the first part of the CYK algorithm, we scan the POS-tags into the bottom row of the chart. We then process each row, from bottom to top, typically in a left-to-right manner, i.e., from starting index 1 to the last possible starting index for that particular span. At each cell we try each possible midpoint, which define the lower-level cells to be used as the children of a new parse constituent in the current cell. We scan through all possible combinations of children to find a two children that occur together on the right-hand side of a production in the input PCFG grammar. In such a way, all the cells in the chart are filled. To improve efficiency, only one entry per non-terminal category is kept in any given cell of the chart, and that is the entry with the highest-probability of all other non-terminals of the same category. The highest-probability parse

[4] Recall from Section 3.1.2, p. 27, that a grammar is in CNF if each of the rule productions in P are either of the form $A \to BC$ or $A \to a$ where $\{A, B, C\} \in V$ and $a \in T$. Any context-free grammar may be transformed to a weakly equivalent grammar in CNF.

```
CYK($w_1 \dots w_n,\ G = (V, T, S^\dagger, P, \rho)$)        ▷ PCFG $G$ must be in CNF

 1  $s \leftarrow 1$                                            ▷ scan in words/POS-tags (span=1)
 2  for $t = 1$ to $n$ do
 3      $x \leftarrow t - 1$
 4      for $j = 1$ to $|V|$ do
 5          $\alpha_j(x, s) \leftarrow b_j(w_t)$
 6  for $s = 2$ to $n$ do                                        ▷ all spans > 1
 7      for $x = 0$ to $n - s$ do
 8          for $i = 1$ to $|V|$ do
 9              $\zeta_i(x, s) \leftarrow \text{argmax}_{m,j,k}\, a_{ijk}\alpha_j(x, m - x)\alpha_k(m, s - m + x)$
10              $\alpha_i(x, s) \leftarrow \max_{m,j,k}\, a_{ijk}\alpha_j(x, m - x)\alpha_k(m, s - m + x)$
```

Figure 3.5: Pseudocode of the CYK algorithm.

can then be traced back from the highest-probability entry with label $S^\dagger$ in the top cell of the chart.

3.2.5 Earley parsing

The CYK algorithm in Figure 3.5 provides an efficient method for CFG parsing. However, with some grammars there may be additional efficiencies to be had. In this section, we discuss a top-down filtering method to try to reduce the number of categories in each chart cell, an idea first presented in the well-known *Earley parsing algorithm* [80].

One way in which the purely bottom-up CYK algorithm can do extra work is in building constituents that cannot subsequently combine with other constituents in a valid parse. If we can recognize that before placing it in the chart, we can avoid wasting any subsequent processing on that constituent. The filtering approach requires a list of categories that are guaranteed not to be dead-ends for any particular cell. This list can be built by taking the closure of the left-hand side/leftmost child relation in the grammar.

Such a filter is typically denoted by *dotted rules*, where the rules that may be used are stored with a dot before the allowed categories for the particular start index. The set of allowable categories are all those which appear immediately to the right of a dot. Once the category to the right of the dot is recognized in a cell with start index x and span s, the dot is advanced to the next word position. If the resulting dot is not at the end of the sequence of right-hand side categories, it is placed in the list of dotted rules for all cells with start index x+s. Note that the filtering approach, and the Earley algorithm that uses it, do not require the grammar to be in this form. The Earley algorithm, as typically presented [80, 202], does binarization on the fly through these dotted rules.

The utility of such top-down filtering is very much dependent on the grammar. There is an overhead to keeping track of the dotted rules, and that overhead may be more than the efficiency gain to be had from the filtering. The Earley algorithm does not provide a worst-case complexity improvement over the CYK algorithm, but under the right circumstances, it can provide much faster performance.

3.2.6 Inside-Outside Algorithm

Just as the CYK algorithm for context-free grammars is analogous to the Viterbi algorithm for finite-state models, there is an algorithm analogous to the Forward-Backward algorithm for context-free grammars, called the *Inside-Outside algorithm* [10, 138].

The *inside* probability for a category $\mathrm{A}_i \in V$ with start index x and span s is the probability of having that category with the particular words in its span:

$$\alpha_i(x, s) \quad = \quad \mathrm{P}(\mathrm{A}_i \overset{*}{\Rightarrow} w_{x+1} \dots w_{x+s}). \tag{3.26}$$

(See Eq. 3.1.2 for a definition of the derives $\overset{*}{\Rightarrow}$ relation.) For non-binary productions with terminals on the right-hand side, the definition is the same as in the CYK algorithm in Figure 3.5:

$$\alpha_j(x, 1) \quad = \quad b_j(w_{x+1}). \tag{3.27}$$

For the remaining binary productions, the inside probability is obtained by replacing the max in the algorithm in Figure 3.5 with a sum:

$$\alpha_i(x, s) \quad = \quad \sum_{m=x+1}^{x+s-1} \sum_{j=1}^{|V|} \sum_{k=1}^{|V|} a_{ijk} \; \alpha_j(x, m-x) \; \alpha_k(m, s-m+x) \tag{3.28}$$

Just like the relationship between the Viterbi and the Forward algorithms presented in Sections 3.2.1 and 3.2.2, the dynamic programming involved in the CYK and the inside part of the Inside-Outside algorithm are identical. Note that, if $\mathrm{A}_r = S^\dagger \in V$, then

$$\mathrm{P}(w_1 \dots w_n) \quad = \quad \alpha_r(0, n). \tag{3.29}$$

The *outside* probability for a category $\mathrm{A}_i \in V$ with start index x and span s is the probability of having all the words in the string up to the start of the constituent, followed by the constituent, followed by the remaining words in the string; i.e., for a string of length n:

$$\beta_j(x, s) \quad = \quad \mathrm{P}(S^\dagger \overset{*}{\Rightarrow} w_1 \dots w_x \mathrm{A}_j w_{x+s+1} \dots w_n). \tag{3.30}$$

We start with the start symbol $\mathrm{A}_r = S^\dagger$ at the root of the tree (top of the chart):

$$\beta_r(0, n) \quad = \quad 1. \tag{3.31}$$

Given a grammar in CNF, any category may participate in a production as either the left child or the right child, so we will sum over both possibilities:

$$\begin{aligned} \beta_j(x, s) \quad = \quad & \sum_{s'=1}^{n-x-s} \sum_{k=1}^{|V|} \sum_{i=1}^{|V|} \alpha_k(x+s, s') \; \beta_i(x, x+s+s') \; a_{ijk} \\ & \sum_{x'=0}^{x-1} \sum_{k=1}^{|V|} \sum_{i=1}^{|V|} \alpha_k(x', x-x') \; \beta_i(x', x+s-x') \; a_{ikj} \end{aligned} \tag{3.32}$$

for a string of length n.

Having calculated the inside (α) and outside (β) probabilities for every category, start index and span, we can now estimate the conditional probability, given the string, of seeing such a constituent in that position:

$$\gamma_i(x, s) \quad = \quad \frac{\alpha_i(x, s)\beta_i(x, s)}{\mathrm{P}(w_1 \dots w_n)} \tag{3.33}$$

We can also calculate the conditional probability of a particular rule production applying at a particular span:

$$\xi_{ijk}(x,s) \quad = \quad \frac{a_{ijk}\beta_i(x,s)}{\mathrm{P}(w_1 \ldots w_n)} \sum_{m=x+1}^{x+s-1} \alpha_j(x,m)\alpha_k(m,x+s) \tag{3.34}$$

These values are analogous to what is calculated for the Forward-Backward algorithm, although instead of time t they are associated with start index x and span s. Just as with the Forward-Backward algorithm, these γ and ξ values can be used to select the parse with maximal conditional probability or to iteratively re-estimate the PCFG model.

3.2.7 Beam Search

During a *beam search* for a solution, some percentage of the paths through the solution space is pruned away because the lower-scoring paths are unlikely to become the best path in the end. In a typical implementation, a threshold value is set and any paths with likelihood smaller than the threshold relative to the most-likely path are pruned away. The threshold can be adjusted dynamically to limit the maximum number of possible (partial) solutions that are under consideration at any time step in the search.

3.2.8 Minimum Bayes Risk Decoding

The Bayes optimal decoding objective is to minimize risk based on the similarity measure used for evaluation. Minimum Bayes risk (MBR) decoding, then, maximizes the expected similarity score of the output solution candidates. MBR decoding for NLP tasks has gained in popularity over the past several years: Jansche [115] used MBR decoding for shallow parsing, Goel et al. [97] used minimum Bayes risk voting strategy for ASR, and Kumar and Byrne [135] performed MBR decoding for translation. DeNero et al. [76] introduce a variant of MBR that can be efficiently applied to a lattice of translations in place of the n-best list comparisons required by traditional MBR decoding, which also has applications in ASR decoding. Some of our work in Chapter 7 (p. 102) approximates MBR inference for parsing.

3.2.9 k-Nearest Neighbors

In *nearest neighbor classification*, new objects are classified by finding the object in the training set that is most similar to the new object, then assigning the category of this "nearest neighbor" to the new object [148]. A simple extension to this method of classification is k-nearest neighbor classification, wherein the k (where k>1) objects in the training set most similar to the new object are consulted for classification. The success of nearest-neighbor classification methods relies heavily upon the *similarity metric* used to compare the classification objects; the "right" similarity metric is sometimes obvious, but sometimes challenging as we discuss in Chapter 7.

3.3 Overview of Application Areas

The next several sections of this chapter will provide a brief background to the areas of tagging, parsing, language modeling, automatic speech recognition (ASR), machine translation (MT), and image classification. Stated very generally, the task in each of these areas is to annotate some type of meta-information on the input data. For parsing, input text is to be annotated with part-of-speech tags, or syntactic trees, or semantic roles, etc. In ASR, the input speech signal is to be annotated with a transcript of what was said. For MT, text input in one language is to be annotated with its translation into another language. In image classification, the input graphical image is to be annotated with labels of any objects captured in the image. Thus, despite the fact that the application areas may seem very disparate, there are actually many similarities at a high level, as will be shown in this section.

3.3.1 Tagging

We have seen several examples already of part-of-speech (POS) tagging tasks, on pp. 4 and 13. In this section we define the objective and evaluation of tagging tasks. Given a word string $w_1 \ldots w_k$ and a tag set $\mathcal{T}$, the tagging task is to find $\hat{C}_1 \ldots \hat{C}_k \in \mathcal{T}^k$ such that:

$$\begin{aligned}
\hat{C}_1 \ldots \hat{C}_k &= \operatorname*{argmax}_{C_1 \ldots C_k \in \mathcal{T}^k} \mathrm{P}(C_1 \ldots C_k | w_1 \ldots w_k) \\
&= \operatorname*{argmax}_{C_1 \ldots C_k \in \mathcal{T}^k} \mathrm{P}(w_1 \ldots w_k | C_1 \ldots C_k)\mathrm{P}(C_1 \ldots C_k) \\
&= \operatorname*{argmax}_{C_1 \ldots C_k \in \mathcal{T}^k} \prod_{i=1}^{k} \mathrm{P}(C_i \mid C_0 \ldots C_{i-1})\mathrm{P}(w_i \mid C_0 \ldots C_i, w_1 \ldots w_{i-1}) \\
&\approx \operatorname*{argmax}_{C_1 \ldots C_k \in \mathcal{T}^k} \prod_{i=1}^{k} \mathrm{P}(C_i \mid C_{i-n} \ldots C_{i-1})\mathrm{P}(w_i \mid C_i). \qquad (3.35)
\end{aligned}$$

Note that this last approximation makes a Markov assumption of order $n+1$ on the sequence of tags, and the assumption that the probability of a word given its tag is conditionally independent of the rest of the words or tags in the sequence.

POS-taggers frequently serve as one of the earlier stages in an NLP pipeline. The parsing pipelines of Collins [63], Bikel [20], and Ratnaparkhi [175] each include POS-tagging stages. The Brill [27] POS-tagger has served as the first stage in many NLP pipelines, including the CoNLL-2000 Chunking Shared Task [207] (discussed in Chapter 5).

POS-tagging is not the only tagging task in NLP; other tasks such as word segmentation [214], shallow parsing [207], named entity recognition [206], and supertagging [13] are oft-implemented tagging tasks. In *shallow parsing*, each word in the input sentence is annotated with its position within a non-hierarchical parse constituent, such as whether the word begins a noun phrase, continues a verb phrase, ends a prepositional phrase, or is "outside" of any such phrase; see Section 5.1 for a more detailed discussion of shallow parsing and its relation to full context-free parsing (reviewed in the next section). Abney [1] set up a parsing pipeline where the first stage shallow parses the input words, then in the second stage, the shallow parse chunks are attached to build the final output parse tree. *Supertagging* is the process of assigning an elementary tree of a Lexicalized Tree-Adjoining Grammar (LTAG) via a tagger to each word of an input sentence; see [118, 119] for a full description of tree-adjoining grammars. Bangalore and Joshi [13] found that by confining their second-stage LTAG parser with output from an upstream supertagger, they were able to prune the parser's search space and thus increase its efficiency by a factor of about 30. Shen and Joshi [197] placed a supertagger as a precursor to an NP chunker (NP-chunking is a subset of shallow parsing). The Clark and Curran [59] combinatory categorial grammar (CCG) parser also uses a supertagger as its first pipeline stage. Birch et al. [23] use CCG supertags to constrain a phrase-based statistical machine translation system (reviewed in Section 3.3.4); Hassan et al. [107] compare the utility of CCG and LTAG supertags in phrase-based machine translation.

Clearly, then, tagging stages are prevalent in NLP systems, both as stand-alone systems [27, 173] and as early stages in pipeline systems. Systems such as these are relevant to this dissertation because their high efficiency and (generally) high accuracy make them attractive for pruning the search space of downstream, higher-complexity stages. In Section 3.10.1 we describe our own finite-state tagger, used extensively throughout this dissertation to conduct empirical trials.

Tagging Evaluation

In order to evaluate a tagged sequence, let us treat the sequence $\boldsymbol{\tau}$ as a set of time indexed tags (τ_j, t), so that $(\tau_j, t) \in \boldsymbol{\tau}$ if the tag for word w_t in $\boldsymbol{\tau}$ is τ_j. Then the intersection of two sequences, $\boldsymbol{\tau} \cap \boldsymbol{\tau}'$, is the number of words for which the two sequences have the same tag. Given the "true" tag sequence $\bar{\boldsymbol{\tau}}$, we can calculate the per-tag accuracy of a given tag sequence $\boldsymbol{\tau}$:

$$\text{per-tag-accuracy}(\boldsymbol{\tau} \mid \bar{\boldsymbol{\tau}}) \quad = \quad \frac{|\boldsymbol{\tau} \cap \bar{\boldsymbol{\tau}}|}{|\mathrm{w}|} \quad = \quad \frac{1}{|\mathrm{w}|} \sum_{(\tau_j, t) \in \boldsymbol{\tau}} \text{tag-in-seq}(\tau_j, t, \bar{\boldsymbol{\tau}}) \tag{3.36}$$

where the denominator in Equation 3.36 is the number of words in the string w, since the true tag sequence must have exactly that length, and where

$$\text{tag-in-seq}(\tau_j, t, \bar{\boldsymbol{\tau}}) \quad = \quad \begin{cases} 1 & \text{if } (\tau_j, t) \in \bar{\boldsymbol{\tau}} \\ 0 & \text{otherwise.} \end{cases} \tag{3.37}$$

Shallow parsing and segmentation tasks may be evaluated either as tag-sequence accuracy using Equation 3.36, or as tree-node F-score using Equation 3.40 (see Section 3.3.2, p. 41).

3.3.2 Parsing

Parsing is an annotation task in which the main objective is to construct a complete syntactic analysis of a string of words. This objective can be achieved through use of a context-free grammar (CFG) (see Section 3.1.2) to provide a model of language processing.

Figure 3.6 shows a simple context-free grammar and the derivable *parse tree* for the input sentence "Stock prices rose in light trading." Note that a parse tree simply represents a particular grammar derivation, and the productions P can be thought of as rules to rewrite symbols into strings of other symbols. Depending on the grammar, there might be many possible parse trees for each input sentence (such as the set of parse trees discussed on p. 5), although for our example CFG there is only one parse for the sentence. The parse tree itself consists of a set of *nodes* such

V = {IN, JJ, NP, NN, NNS, PP, S, VP, VBD}
$S^\dagger$ = {S}
T = {in, light, prices, rose, Stock, trading}

$$P \quad = \quad \left\{ \begin{array}{l} \text{IN} \rightarrow \text{in}, \\ \text{JJ} \rightarrow \text{light}, \\ \text{NN} \rightarrow \text{Stock}, \\ \text{NN} \rightarrow \text{trading}, \\ \text{NNS} \rightarrow \text{prices}, \\ \text{NP} \rightarrow \text{JJ NN}, \\ \text{NP} \rightarrow \text{NN NNS}, \\ \text{PP} \rightarrow \text{IN NP}, \\ \text{S} \rightarrow \text{NP VP}, \\ \text{VBD} \rightarrow \text{rose}, \\ \text{VP} \rightarrow \text{VBD PP} \end{array} \right\}$$

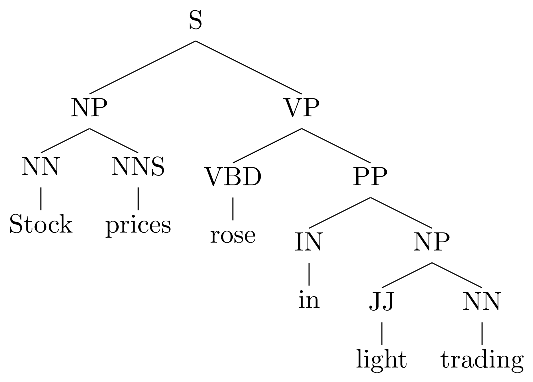


Figure 3.6: An example context-free grammar and derived parse tree.

as the S sentence node, the NP noun phrase nodes, and the VP verb phrase nodes in the figure, which may also be referred to as *parse constituents.* The relations between nodes in a tree are often discussed in terms of *parent* and *children* nodes, where each node has only one parent but may have n children; the VP node in the figure has two children (VBD and PP), and the S node is its parent. Note that a parent node occurs on the left-hand side of a CFG rule production, and the children nodes are on the right-hand side of the production. The "top" of a parse tree (the S node in the figure) is referred to as its root; terminal symbols (words) are at the leaves of the tree; and pre-terminal POS symbols (IN, JJ, NN, NNS, and VBD) are one level higher than the terminal nodes.

The probability distribution for a PCFG can be estimated in several different ways; one of the more common is via relative frequency estimation or some other supervised learning method, estimating the PCFG distribution from training data provided by a *treebank.* A treebank consists of sentences that have been human-annotated with syntactic structure represented as a tree; typically the annotation is performed by linguists and follows detailed annotation guidelines. The Penn Wall Street Journal (WSJ) Treebank [150], consisting of one million words from the Wall Street Journal, will be referred to extensively throughout this dissertation. Other examples of treebanks include the SWITCHBOARD corpus [96], the Brown corpus [89], the Prague Dependency Treebank [101], and the Rhetorical Structure Treebank [36].

Finite-State, Context-Free, and Context-Sensitive Parsers

Human language is known to be beyond context-free, and thus context-free grammars represent an approximation of human language. Approximations are, in fact, quite common as we saw in Section 3.3.1 when we discussed finite-state shallow parsers. Balfourier et al. [11] present a parsing system capable of producing different levels of syntactic annotation, from finite-state to context-free, based on an input tuning parameter. Other parsers go beyond context-free to implement context-sensitive grammars, such as tree-adjoining grammars (TAGs) [118, 119] and unification grammars [123]. It has been shown that the CYK and Earley algorithms can be extended for TAGs and CCGs, to a polynomial algorithm of $O(n^6)$; this high level of complexity generally prevents context-sensitive grammars from becoming commonly used.

Figure 3.7 shows the Ratnaparkhi [175] parsing pipeline as an example parsing pipeline, where a parse tree is essentially built in layers, beginning with the POS-tags, then the shallow chunks, then finally the full context-free tree. Context-free parsers have been implemented as pipelines, and as stages in the middle or at the end of a pipeline. Parsing is often promoted as a beneficial pre-cursor to other tasks such as information retrieval, summarization, even machine translation. The empirical trials conducted throughout this dissertation focus on understanding and improving context-free parsing pipelines, specifically the Charniak and Johnson [44] pipeline (discussed in detail in Section 3.10.2).

Parsing Evaluation

In context-free parsing, parse candidates are evaluated according to the F_1 metric, which is the harmonic mean of *recall* and *precision* (Equations 3.38 and 3.39, respectively); recall is the number of tree nodes in the truth that were identified by the system, while precision is the number of tree nodes identified by the system that were in the true tree. For evaluation of a parse tree A, the

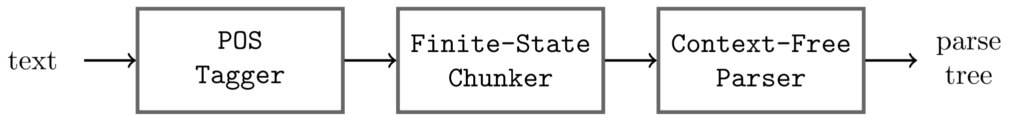


Figure 3.7: An example parsing pipeline.

parse is compared to the true reference parse B as follows:

$$R = \frac{|A \cap B|}{|A|} \tag{3.38}$$

$$P = \frac{|A \cap B|}{|B|} \tag{3.39}$$

$$\begin{aligned} F_1 &= \frac{2 * R * P}{R + P} \\ &= \frac{2 * |A \cap B|}{|A| + |B|}. \end{aligned} \tag{3.40}$$

A correct parse, i.e., one which is identical to the gold-standard parse, has an F-score of 1.0. Evaluation is performed using the `evalb`[5] scoring script, which operates on the set of labeled brackets that represent the parse tree. Both the span and the label of the brackets are included in the evaluation. Standard ParsEval [24] parameterizations ignore some node differences for evaluation, including: different POS tags, ADVP/PRT substitutions, and disagreement on bracket boundaries that span punctuation marks.

3.3.3 Automatic Speech Recognition

The objective of automatic speech recognition (ASR) is to begin from an observed acoustic signal, and end with a transcription of the possible word hypotheses inferred from the signal. A simple example of speech recognition is an isolated word recognition task, in which the ASR system discriminates between two possible words such as "yes" and "no." Other, more complex tasks might be to automatically transcribe a television newscast to provide closed captions for the telecast; recognize vocal commands (e.g., "Call Jane," or "Find the nearest Mexican restaurant"); transcribe voicemail to e-mail or dictate an office memo. Tasks such as these can require large vocabulary continuous speech recognition (LVCSR).

LVCSR provides a probabilistic estimate of the sequences of words at the source of an unsegmented audio stream. Figure 3.8 shows a typical ASR pipeline, using HMMs with a beam search approximate inference algorithm. The system uses three recognition models: an acoustic model, a pronunciation model, and a language model. The *acoustic model* is a model for an elementary unit of speech such as a phone, typically represented as a three-state finite-state transducer with self-loops at each state. Acoustic models in state-of-the-art LVCSR systems are usually context-dependent models, modeling up to 3-, 4-, or 5-phone sequences [140, 211]. The *pronunciation model* defines word-level units, as a concatenation of the acoustic models for each phoneme of the word. Most LVCSR systems have a pronunciation dictionary comprising hundreds of thousands of words [52, 141], some of which may have multiple possible pronunciations. See the well-known CMU Pronunciation Dictionary[6] for an example of such a model. The *language model* constrains the possible words sequencing. Formally, the probability of a transcribed word sequence w given the audio input a is typically calculated using the noisy channel model, which has also been used in SMT (Section 3.3.4):

$$\begin{aligned} \underset{w}{\operatorname{argmax}} \mathrm{P}(w|a) &= \underset{w}{\operatorname{argmax}} \frac{\mathrm{P}(a|w)\ \mathrm{P}(w)}{\mathrm{P}(a)} \\ &= \underset{w}{\operatorname{argmax}} \mathrm{P}(a|w)\ \mathrm{P}(w) \end{aligned} \tag{3.41}$$

where $\mathrm{P}(w)$ is the language model and $\mathrm{P}(a|w)$ is the acoustic model.

[5] http://nlp.cs.nyu.edu/evalb/

[6] http://www.speech.cs.cmu.edu/cgi-bin/cmudict

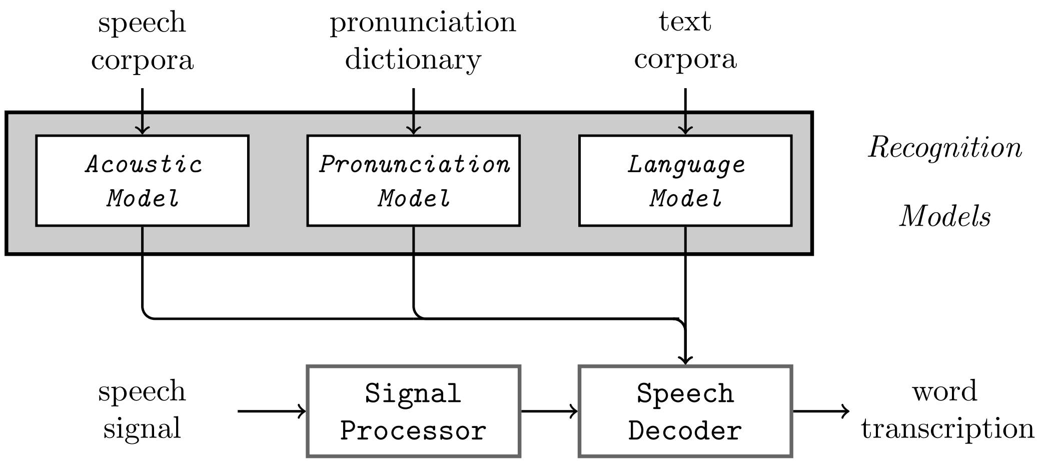


Figure 3.8: An example LVCSR pipeline.

The search space of an LVCSR decoder is represented by all possible word sequences given the features extracted from the input audio stream. To conceptualize the size of such a search space, imagine a 50,000 word vocabulary system (small by today's state-of-the-art standards). A typical word has 6-7 phonemes, each of which consists of the three HMM stats from the acoustic model. Thus an exact inference search would have to search over $50,000{*}7{*}3$ = one million states for each recognized word, making the search computationally challenging if not impossible.

Spectral-based speech features are extracted by performing signal-processing on the audio input; Figure 3.9 shows an example of the spectral features for the word "weekend." The speech decoder then computes the most likely word sequence based on the extracted speech features and the recognition models. The naïve approach would be to explore and compute the probability of all possible sequences of words and select the most probable one. With over a million states for each word, it is easy to see why most LVCSR systems implement beam search [114] or other pruning strategies in decoding, including implementing a pipeline architecture.

The recognition models are independently trained and compiled offline, but applied in a pipeline manner during testing. The pipeline-final output word sequence is constructed by a concatenation of phone-units to form word-units, which in turn are concatenated to form word sequences. Beam search is typically employed at each of these models during testing, and if the phone-units required to construct a particular word are pruned away in the acoustic model's beam search, then clearly the language model will be prevented from selecting that word as output.

ASR Evaluation

The accuracy of an end-to-end speech recognition system is typically measured in terms of word error rate (WER). WER is calculated as:

$$\text{WER} = \frac{S + D + I}{N} \tag{3.42}$$

where S is the number of substitutions made by the system, D is the number of deletions, I is the number of insertions, and N is the number of words in the reference. Values of WER closer to 0 indicate higher accuracy. Other measures of accuracy include Single Word Error Rate (SWER) and Command Success Rate (CSR).

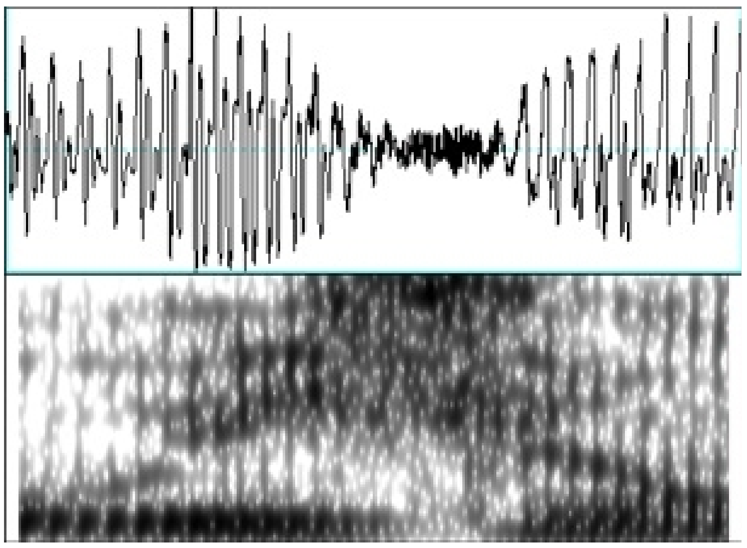


Figure 3.9: Waveform (top) and spectrogram (bottom) of the word "weekend."

3.3.4 Machine Translation

The objective of machine translation (MT) is to automatically map from a *source* language string to a *target* language string. We discussed a partial example in Chapter 1, translating our example English sentence "Stock prices rose in light trading" into a Spanish sentence "Precios de las acciones subían en intercambio libiano." Ideally, machine translation would be conducted by mapping the source sentence to some *semantic interlingua* representation, e.g. "rise(price([plural],[stock]), trade([gerund],[light]), past)," then synthesizing a sentence in the target language from the interlingua representation. However, the current-best translation systems are data-driven statistical models that rely on surface representations of the strings such as lexical identity and syntactic function. The basic approach behind a statistical machine translation (SMT) system follows the noisy-channel approach from automatic speech recognition (ASR) (see Section 3.3.3). The probability of translating a source string f into a target string e is thus calculated as:

$$\begin{aligned} \operatorname*{argmax}_e \mathrm{P}(e|f) &= \operatorname*{argmax}_e \frac{\mathrm{P}(f|e)\ \mathrm{P}(e)}{\mathrm{P}(f)} \\ &= \operatorname*{argmax}_e \mathrm{P}(f|e)\ \mathrm{P}(e) \end{aligned} \tag{3.43}$$

using Bayes rule.

In Equation 3.44, $\mathrm{P}(f|e)$ is termed the *translation model* (similar to the acoustic model in ASR; see Section 3.3.3 for more details) and $\mathrm{P}(e)$ is the *language model* (discussed previously, in Section 3.3.3). In the example MT pipeline in Figure 3.10, the Lexical Translation and Phrasal Translation stages each contribute to $\mathrm{P}(f|e)$, and the Language Model stage provides $\mathrm{P}(e)$ to the decoding stage. Translation models assigns a probability to a pair of strings $<f,e>$ such that if f "looks like" a good translation of e, then $\mathrm{P}(f|e)$ will be high. The translation pairs $<f,e>$ would ideally come from human translations, but again, human-produced data is expensive and slow. Therefore, translation-pairs of sentences are extracted for MT from *parallel corpora*, document pairs that are translations of each other such as the European Parliamentary proceedings[7] [130], required by law to be published in 11 different languages.

[7] http://www.statmt.org/europarl/

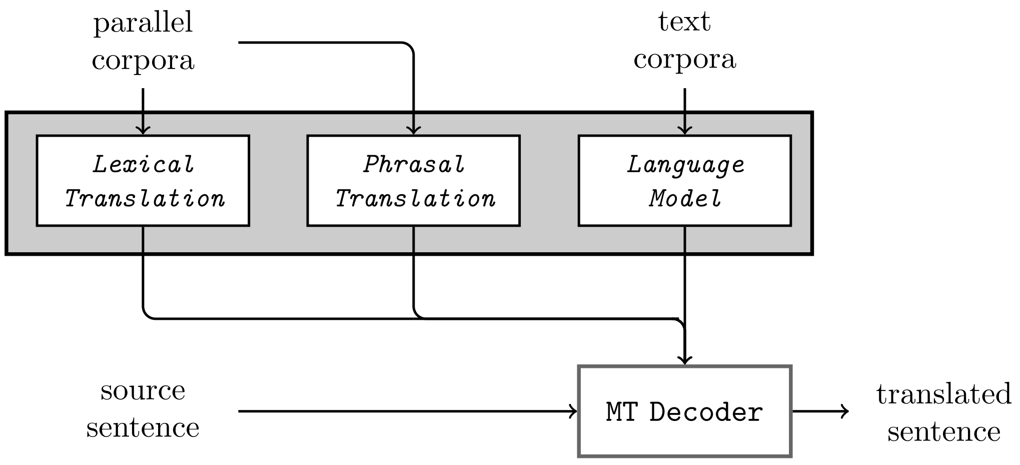


Figure 3.10: An example phrase-based MT pipeline.

Comparable corpora have also been used in machine translation [196, 205]. Comparable corpora are texts in two languages that are similar in content, but are not translations. One example is a set of news articles published on the same day in different countries (and languages).

Lexical and Phrasal Translation

To extract the necessary information from document-aligned parallel corpora, each document is first sentence-aligned, typically using simple heuristics based on sentence length, such as the Gale and Church [92] algorithm. Then the words within each aligned sentence are also aligned using various algorithms [30, 164, 155]. Word-alignment quality can be intrinsically measured by alignment error rate (AER) [164]:

$$\text{AER}(S, P; A) = 1 - \frac{|A \cap S| + |A \cap P|}{|A| + |S|}. \tag{3.44}$$

Here a recall error occurs if a "sure" S alignment link is not found in A, and a precision error occurs if an alignment link in A is not in the "probable" set of alignment links P. Ayan and Dorr [7] demonstrated that AER does not correlate well with translation quality, but we include its definition here for completeness.

Phrasal translation translates sequences of source-language words into sequences of target-language words. Some of the advantages of phrase-based translation are: many-to-many alignments from source to target are permitted, as opposed to word-based translation; and the phrases allow for more context in translation than would be provided by a word-based translation system. The key component in phrasal translation is the phrase table, which is *extracted from* the word alignments, essentially by "growing" the alignments diagonally to form aligned phrases. The gray cells in Figure 3.11(b) represent a few of the phrases that could be extracted from the word alignments in Figure 3.11(a).

MT Evaluation

Translation quality would, ideally, be evaluated by a human to determine *adequacy* (how well the translation captures the information of the source sentence) and *fluency* (whether the translation is a "good" sentence in the target language). However, human evaluation is slow and expensive, so translation quality is measured by the precision-based Bleu metric [168], where the quality of a translation is considered to be how closely the translation output by a machine matches a

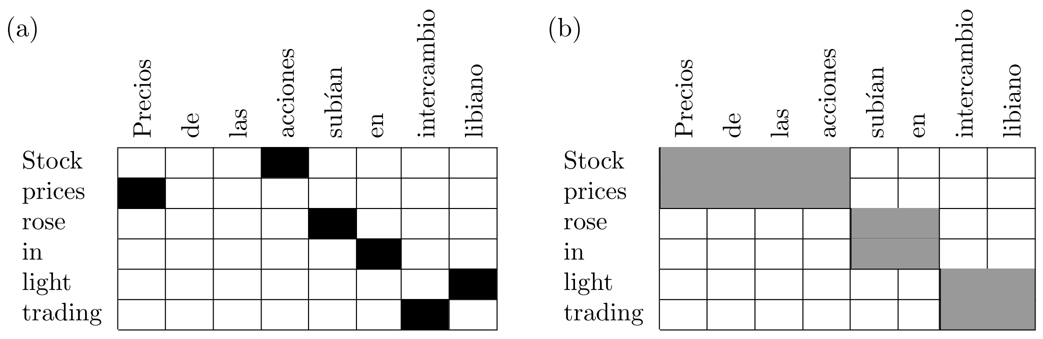


Figure 3.11: An example word alignment (a) in tabular format, and a few of the phrases (b) to be extracted based on the word alignment.

professional human translation.

Bleu is calculated as the geometric average of the modified n-gram precisions, p_n, using n-grams up to length N:[8]

$$\log \text{Bleu} = \min(1 - \frac{r}{t}, 0) + \sum_{n=1}^{N} \frac{1}{N} \log(p_n) \tag{3.45}$$

where the first factor represents a brevity penalty, and p_n is the modified n-gram metric.[9] Values of Bleu range from [0..1], where values closer to 1 indicate candidate translations that overlap more with, and thus are more similar to, the reference translations.

3.3.5 Image Classification

In image classification,[10] information is extracted from an input image, such as the identity and label of any objects in the image. The image data may take many forms, such as photographs, video sequences, or multi-dimensional medical images from a CAT scan. The task of determining whether or not an image contains some specific object, feature, or activity can normally be solved fairly easily by a human, but identifying arbitrary objects in arbitrary situations remains difficult for computers. Much like in ASR and parsing systems, existing methods for image classification excel by limiting the domain of the problem: only recognizing specific objects, such as simple geometric shapes, human faces, or hand-written characters. Further restrictions might also restrict the background or pose of the object to be recognized.

Image recognition can take on several subclasses: recognizing a specific object or class of objects; identifying an individual instance of an object; or detecting specific conditions in the image. In object recognition, systems are trained to recognize specific classes of objects (such as planes, buildings, or bodies of water) along with the position of the object in the image or scene. In object identification, a specific object such as a specific person's face or fingerprint is identified. In image detection, an image is scanned to determine whether or not a particular condition is present, such as detecting whether a car is present in an image of a road.

Detection tasks are relatively simple and fast, so detection may serve as the first stage in an image classification pipeline, to find smaller regions of interesting image data to be further analyzed

[8] N=4 was found to correlate best with human judgements.

[9] Bleu is a *modified* precision metric because the count of n-gram matches between a candidate translation T and a reference translation R is modified such that n-grams in the reference are matched only once to an n-gram in the candidate translation.

[10] Image classification may also be referred to as object recognition, image processing, or image analysis, and is a subfield of computer vision research.

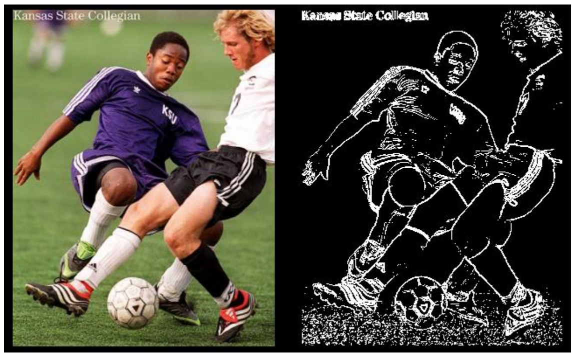


Figure 3.12: Edges detected (right) from original input image (left).

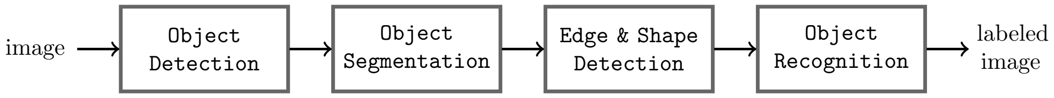


Figure 3.13: A simple image classification pipeline.

by more computationally demanding techniques later in the pipeline. Early stages in an image classification pipeline might also perform local operations on the image such as *edge detection* or noise removal. The goal of edge detection is to determine the edge of shapes in a picture and to draw a resulting bitmap where edges are in white on black background; edge detection is typically performed by simple bitwise comparisons, and so is very fast and relies only on local information in the image. Detecting the edges of shapes provides a starting point for downstream processing on those shapes. Figure 3.13 shows an example image classification pipeline.

Optical character recognition (OCR) is a common image classification problem, in which the objective is to identify a printed or handwritten letter or digit in the input image, usually with the goal of encoding the printed or handwritten text into ASCII format; such tasks have also benefited from the use of a language model. Many image classification systems approach the problem as one of pattern recognition, matching the input image against a set of reference, labeled images, and thus are based on the nearest-neighbor search. The nearest-neighbor search requires some measure of similarity between two data points, which can be computationally expensive to calculate; therefore, much attention has been given to finding efficient and/or approximate similarity metrics for comparing images.

Image Classification Evaluation

Image classification systems are often evaluated by simple recognition accuracy R (or by classification error rate, calculated as $1-R$):

$$R = \frac{c}{t} \tag{3.46}$$

where c is the number of correctly recognized objects (or correctly classified images) and t is the total number of images input for analysis. Image recognition may also be evaluated by plotting a system's false-positive rate against its true-positive rate and calculating the area under the resulting Receiver Operating Characteristic (ROC) curve [219]; an area value of 1 would indicate an errorless system.

3.4 Pipeline Terminology

One of the goals of this dissertation was to identify commonalities among pipeline systems across several different fields, which is difficult due to the lack of a shared vocabulary to describe pipeline systems across—and even within—research communities. Thus in this section we briefly discuss some of the more common terms used to describe pipeline systems in the existing literature, and map these terms to our pipeline framework vocabulary.

Pipeline systems have often been referred to as "**cascaded**" systems (e.g., [1, 2, 6, 31, 83, 72, 73]); these systems are typically feed-forward, single-source, partial-solution constrained pipelines (Section 3.5). "**Coarse-to-fine**" systems (e.g., [45, 44, 74, 131]) are typically feed-forward, single-source, complete-solution constrained pipelines (Section 3.6) with an emphasis on systematically reducing the search space and increasing the complexity of downstream models. "**Ensemble**" systems [77, 78, 167, 176, 191] are also feed-forward, complete-solution constrained pipelines, typically utilizing multi-source constraints for a downstream voting stage (Section 3.6.3). The term "**multi-pass**" has two different senses; the most common sense refers to the multiple stages of a pipeline systems (e.g., [43, 74, 99, 208, 210]), and the other sense refers to a complete pass through a system as in iterative pipelines (Section 3.8.2). The "**multi-layer**" architecture in [90] (not to be confused with a multi-layer perceptron), the "**multi-level**" tags in the McDonald et al. [154] dependency parser, and the "**multi-scale**" image processing algorithms [81, 131] can be interpreted as references to the multiple *stages* of a pipeline architecture.

In general, pipeline systems can be identified by the presence of multiple processing stages where the output of one (or more) stages defines the input of the next stage in the pipeline: exactly as stated in our definition of pipelines. The next several sections of this chapter will classify many existing pipeline systems according to our pipeline framework in Chapter 2.

3.5 Partial-Solution Constrained Pipelines

The first, and easiest, distinction that we make is between the two classes of partial-solution and complete-solution constrained pipelines. In a *partial-solution constrained pipeline*, the constraints passed from one stage to the next are partial constraints, and will be extended or added to for the pipeline-final output. In a *complete-solution constrained pipeline*, the constraints fully specify the search space of the next stage. We could also make a distinction between one-best pipelines and n-best pipelines; in n-best pipelines, a *set* of candidates (either partial- or complete-solutions) are output from one stage for input to the next stage in the pipeline. That set of candidates may either function as complete-solution constraints for the next stage, or as a disjoint set of partial-solution constraints; Curran et al. [71] showed the benefits of using disjoint sets of partial-solution constraints to maintain some level of ambiguity in their CCG parsing pipeline. In the case of a one-best pipeline, the single candidate passed from one stage to the next must be a partial-solution constraint.

The Ratnaparkhi [175] parser is a classic partial-solution constrained parsing pipeline. The first stage generates POS tags over the input sentence, the second stage generates chunks over the POS-tagged sentence, and the third stage iteratively joins the chunks into a hierarchical tree. Input to the second stage consists of n-best POS-tag sequences, and input to the third stage consists of n-best distinct POS-and-chunk sequences; each stage was trained on gold reference data.

Many other parsing pipelines make use of partial-solution constraints. The Abney [1, 2] "partial parser" parses a sentence by first chunking the words in the sentence then using an attacher to attach the chunks into a final parse tree. Many edge detection systems [18, 131, 147, 204] function in a similar fashion, by detecting part of an edge at each stage, to be extended by the next stage.

In the Charniak et al. [45] "multi-level" parsing pipeline, the first stage produces a high-recall unlabeled parse tree for the input sentence, and the downstream stages each successively relabel the parse tree output by the upstream stage, with label sets increasing in size (and level of detail) at each stage. The so-called "two-pass" architecture presented by Charniak and Johnson [43] detects and removes edited words in transcribed speech before passing the transcription to the downstream parser. In the Joshi and Bangalore [117] supertagging pipeline, an initial stage selects a supertag for each word in the input sentence, and the next stage links the sequence of supertags to form a dependency parse. Bangalore [12] extended this pipeline to use supertag sequences as input to a language modeling stage, and demonstrated that although supertagging is a more difficult task than POS-tagging, the increased level of detail provided by supertags over POS-tags led to an increase in performance in the downstream language model. The supertagger implemented by Bangalore and Joshi [13] contains three stages: a POS-tagger (implemented by Church [53]) followed by a supertagger which used the input POS tags as a back-off model for unknown words, and finally a parser. The article focused on improving parser efficiency by pruning the parser's search space with supertag-constraints, and reported oracle F-scores in the low 90s using the three-best supertag sequences. Clark and Curran [56] present an interesting approach to partial-solution constraints, training a dependency parser on only partially-annotated training data.

Parser output has also been used to define partial-solution constraints for machine translation pipelines. Bangalore and Riccardi [15] use output from an upstream dependency parser to perform lexical re-ordering, then extended this notion of separating translation from re-ordering in [16] by performing lexical choice at the first stage and outputting the lexical translations to the second stage for lexical reordering. Fox [88] and Cherry and Lin [48] both use output from an upstream dependency parser to disallow crossing dependencies in machine translation candidates. Charniak et al. [46] use a parser as a pipeline-final stage to prune unlikely partial translations output by an upstream translation model. Collins et al. [67] translate by parsing the source input sentence in the first stage of their pipeline, performing a series of transforms on the parse tree in the second stage to re-order the underlying sentence, then translating the re-ordered source sentence in the final stage. Sporleder and Lapata [201] examined the utility of discourse chunking as an upstream stage for sentence compression.

Several pipeline systems implement upstream stages intended specifically to improve efficiency in downstream stages, including our own work [181, 182] using chart constraints to reduce the observed-time complexity of a context-free parser (see Section 3.10.3 for more details about this work). DeNero et al. [74] implement similar steps to reduce the observed-time complexity of their transduction grammars for machine translation. Birch et al. [22] propose constraining the search space of the Joint Probability Model (proposed by Marcu and Wong [149]) phrase-extraction to only those phrases supported by the word alignments output by an upstream stage. Such pipelines are somewhat unusual because they are typically evaluated extrinsically (i.e., for efficiency gains) rather than intrinsically (for mid-pipeline accuracy improvements).

3.5.1 Model Refinement

Most of the pipelines discussed in this dissertation pass solutions or partial solutions from stage to stage. However, some of the pipelines pass information regarding model parameters rather than (or in addition to) solution sets.

In the seminal IBM word-alignment pipeline [30, 29], each stage provides initial estimates for the model parameters of the next stage. In this pipeline, IBM Models 1-5 each constitute one stage of the pipeline, and each stage outputs a single partial solution for the downstream stage. Each model calculates the conditional probability $\Pr(f|e)$, the likelihood of the French sentence f as a translation of the English sentence e, with each model in the pipeline adding more conditioning information to the probability estimation than was used in the upstream model, following a typical increasing model-complexity pipeline design. In this pipeline system, in addition to providing a partial solution constraints, upstream stages are also used to initialize the parameters of the next stage. GIZA++ [164] extends the IBM models for word alignment, adding a Model 6 and replacing Model 3 with an HMM model, and is similarly an example of both model- and solution-refinement in a pipeline system.

In the Collins [63] parser, parsing Models 1, 2, and 3 are similarly trained under a pipelined model-refinement scenario [20]. Note that the Collins parser requires POS tags[11] on input at test-time, creating a more traditional solution-refinement pipeline. However, the probability distributions of the parser are estimated in training from reference POS tags, only relying on the input POS tags to provide back-off estimations for unknown words; thus the POS-tagging stage only participates in the pipeline at test-time.

3.5.2 Soft Constraints

Hard constraints define the pipeline architecture, and thus the majority of systems analyzed in this dissertation utilize hard constraints. However, soft constraints also have a role in pipeline systems. Cherry and Lin [49] use the output of a bi-text synchronous parser [212] to impose soft constraints on downstream word alignment. In their paper, Cherry and Lin note that implementing soft constraints in a system using a beam search is problematic, because the imperfect and incomplete search conducted in a beam search scenario might not be able to find the optimal solution under the soft constraints; they argue that exact inference [125] is a better search strategy with soft constraints.

Many systems use the model score of an upstream stage, which can be thought of as providing a soft constraint. The `Hiero` machine translation system [50, 51] uses the score output by an upstream language model as a feature in its log-linear machine translation decoder. The Charniak and Johnson [44] final-stage reranker uses the probability scores output by the upstream parser as a feature.

3.6 Complete-Solution Constrained Pipelines

A complete-solution constrained pipeline can most easily be identified by the presence of a *ranking* stage in the pipeline, wherein the solution candidates input to the ranking stage are scored and ranked. The score of each candidate may be calculated as a sum of "votes" from upstream stages, or recalculated internally by the ranking stage, as is the case for a MaxEnt reranking stage, for example. The output of the ranking stage may be the full set of input candidates in some rank order, or it may simply be the top-ranked solution candidate.

Complete-solution constraints are most commonly represented as n-best lists, as we will see below, but can also be represented in a condensed format such as a word lattice or chart. The Charniak and Johnson [44] pipeline, discussed at length in Section 3.10.2, is an exemplary complete-solution constrained pipeline; a condensed parse forest (chart) is passed from the first parsing stage to the second, and n-best lists of parses are passed from the second parsing stage to the final, reranker stage.[12] May and Knight [151] examined the different effects of these two representations by constructing a lattice from an input n-best list. Huang and Chiang [113] presented results of rescoring a "forest" of parse trees as opposed to a list of parse trees.

[11] Results reported in [63] used the Brill [27] POS-tagger.

[12] We also implemented this model as a *ranking* stage, where the the entire set of candidates was output and the model was optimized for rank-accuracy; see p. 56 for details.

3.6.1 Ranking

Shen and Joshi [198] examine the differences between *selection* and *ranking* systems. They focus on the use of large margin machine learning techniques; in selection stages, the goal is to find the margin, or hyperplane, that will separate the best solution candidate from the others in the space. They demonstrated the use of uneven margins in ranking, where the goal is to find larger margins to separate higher-ranked candidates and to allow smaller margins for separating the lower-ranked candidates. The PRanking system implemented by Crammer and Singer [70] is another exemplary ranking system, in which each input data point is assigned a rank (or rating); the model is evaluated based on rank-accuracy.

3.6.2 Reranking

Reranking stages,[13] which select from an n-best list or lattice on input, are *complete-solution constrained* pipelines. In a typical reranking stage, as discussed in Chapter 2 (p. 19), a list of the n-best solution candidates produced by a baseline system is re-scored, typically using richer, more complex models, in order to select a better candidate from among those provided.

Reranking stages have been incorporated into many NLP pipelines, including other parsing pipelines [64, 68, 69], speech recognition [183], and machine translation [166]. Hall et al. [103] explored several different methods for reranking dependency parses. Shen et al. [199] created one of the very few reranking systems for machine translation, which takes as input an n-best lists of translations. In order to see even an insignificant gain in accuracy, the size of the input n-best lists were an order of magnitude larger than those used in reranking parser output. DeNero et al. [76] apply a variant of minimum Bayes risk decoding to re-score n-best lists and lattices of translation candidates. Bannard and Callison-Burch [17] rerank paraphrases extracted from parallel corpora, using a language model as the reranking stage. Callison-Burch [33] extended this pipeline by adding a parsing stage at the beginning of the pipeline and filtering proposed paraphrase-pairs to only those with the same syntactic function. In the `Julius` speech recognition system [139], the first stage performs a beam search, and the second stage reranks the recognition results from the first stage using a more-complex language model and a context-dependency model. Roark et al. [186, 184, 185] use a discriminatively-trained language model to rerank the output of an LVCSR system.

3.6.3 Voting

In a voting pipeline, each of the upstream stages[14] "vote on" each of the input candidates. The candidate with the largest number of votes is then the best candidate and is output. The voting method might be a simple majority vote, where the candidate output from the majority of the upstream stages is selected for output. It may also be a score combination method, where the scores generated by each upstream stage for each of the input candidates are combined and the candidate with the highest combined score is selected for output. The ASR pipeline systems implemented by Fiscus [84], Goel et al. [97], and Sagae and Lavie [189], are all exemplary *voting* pipelines. Mintz [156] uses voted consensus for edge detection in image analysis. Bangalore et al. [14] use voting techniques to create cleaner training data for machine translation systems. Henderson and Brill

[13] As Shen and Joshi [198] noted, the term "reranking" here is something of a misnomer, in that the pipeline stages discussed herein are actually *selection* stages rather than *ranking* stages, according to the definitions from our pipeline framework, but "reranking" has become the conventional term used in NLP.

[14] Voting pipelines are often also instances of multi-source pipelines, as discussed further in Section 3.7.

[108] combine parsers to "exploit diversity." Buchholz et al. [31] found that adding chunkers—even lower-performing ones—as upstream voters improved the performance of their final-stage grammatical-relations finder. Thus this paper is one of only a few in the literature which support the notion that poor upstream performance does not necessarily correspond to poor downstream performance, a notion that will be explored further in Chapter 6.

The composition of weighted finite-state transducers in certain semirings can be thought of as voting pipelines, and are used in many different application areas including parsing [115], speech recognition [158, 169], language modeling [12], and machine translation [3, 127, 134, 136, 151, 163].

Recombination

A subset of voting pipelines are those that utilize *recombination* techniques. In a recombination pipeline, the complete-solution constraints are broken down into partial solutions and "re-combined" to create a solution that may not have been output by any of the upstream stages. For example, in the Sagae and Lavie [189] parsing pipeline, the input parse trees are broken down to the individual nodes in each tree, then the set of nodes are re-combined into a parse tree. Riezler and Maxwell III [180] similarly re-combined parse constituents based on constituent similarity to produce a parse tree. The Goel et al. [97] ASR pipeline is also a recombination pipeline; it takes as input a set of transcriptions (word sequences), and conjoins the sequences to create a word lattice, then the best path through this lattice is output from the pipeline-final re-combination stage. The pipeline systems implemented by Liang et al. [145] and Klementiev et al. [126] are also examples of recombination pipelines.

3.6.4 Filtering

The cascade of classifiers presented in Athitsos et al. [6] is an interesting case. Each input data point (in this case, an image to be classified according to its nearest neighbors) is *not* pushed through the entire sequence of classification stages. Rather, if any stage classifies the image within a thresholded level of certainty, then the classification at that stage is used as the "pipeline-final" solution for the given image. Any data points that reach the last stage in the pipeline are classified by that stage regardless of the level of certainty. Certainty thresholds are set for each stage during training to maximize performance (where performance is measured both in terms of accuracy and efficiency). This technique of filtering data points at each stage in the pipeline appears to be unlike any of the pipeline techniques implemented in parsing, MT, or ASR systems; some of our future work (Chapter 9) is to explore the applicability and utility of this technique in other pipeline systems.

3.7 Multi-Source Pipelines

Multi-source, partial-solution constraints often consist of heterogeneous output from several different upstream stages. For example, Schafer and Yarowsky [193] use the output of an upstream POS-tagger and a shallow parser to filter the translation candidates proposed by their machine translation system.

3.7.1 Homogenous Sources

Co-trained and self-trained pipelines are examples of homogenous multi-source pipelines. Clark et al. [60] bootstrap POS taggers using co-training, where two models are iteratively[15] trained

[15] Co-trained and self-trained models are also iterative pipelines, discussed in Section 3.8.2.

on the output of the other. Bootstrapping and domain adaptation such as that performed by Bacchiani et al. [8] are also examples of homogenous multi-source pipelines. Fink and Perona [82] use mutual boosting for image classification.

3.7.2 Heterogeneous Sources

The Moses machine translation pipeline [129], an open-source implementation of the Pharaoh [128] translation system, is a canonical example of a multi-source pipeline. Moses uses the output of a word-alignment system (such as GIZA++ [164], which is itself a pipeline system) as the first stage in a phrase-based translation pipeline. This system extracts phrases—or more accurately, sequences of co-located words—from the output of a word-alignment system using a variety of heuristic techniques, and phrases are then translated as a unit. The Moses decoder uses phrasal and lexical translation scores as well as language model scores for the output translation.

Multi-source pipelines are fairly common. Cohn et al. [62] train different conditional random field models on subsets of a given task—for example, training a separate model for each of 23 shallow parsing tags [207]—then combine the models to produce the pipeline-final output. Since each of the models are trained for different sub-tasks, the output of the models are heterogeneous. Similarly, Punyakanok and Roth [171] combine output from several different classifiers such that the combined output satisfies some given inference constraints. "Opinion-pool" systems [200, 203] train different feature sets separately on the same dataset, then the feature sets are combined in the pipeline-final stage. Kim et al. [124] present a multi-source system to detect and remove disfluencies by passing lexical and prosodic information through two initial stages: a decision tree stage and a language model stage; the output from these two stages is then input to a transformation-based learning stage. The speech recognition pipelines of SPHINX [140], HTK [211], and Chong et al. [52] all use output from an acoustic model (or models), a pronunciation model, a language model, and a signal processing stage as input to the pipeline-final decoding stage. Vergyri [209] also explored the use of multiple knowledge sources to improve speech recognition.

Sampling pipelines such as those presented by Arun et al. [5], DeNero et al. [75], Finkel et al. [83] may also be thought of as multi-source pipelines.

3.8 Pipelines with Feedback and Iteration

3.8.1 Feedback Pipelines

The C&C Combinatory Categorical Grammar (CCG) parser [56] is a rare example in NLP of *feedback* in a partial-solution constrained pipeline. The first stage, a supertagger, calculates the posterior probability of a set of supertags for each word in an input sentence. The supertagged words are passed on to the next stage, a CCG parser, which combines the supertags to create a parse spanning the entire input sentence. The number of supertags per word, passed from the first stage to the second, is controlled by a parameter α. During training, the set of supertags for each word is forced to include the correct supertag, which may require a manipulation of the output; the manipulation does not, of course, occur during testing. At test-time, if the parser is unable to combine the tags in such a way as to cover the entire sentence, then the parser can go back to the supertagger (up to five times) to request more supertags. The system was designed such that the parser can request more supertags for a particular word or simply a change in the α parameter; focusing the altered search on particular words resulted in fewer feedback requests but came with a higher risk in that expanding the search space around that particular word could still lead to a failure to find a global solution. The feedback mechanism is triggered on failure in the parser.

Note that the limitation of five rounds of feedback means that the parser can (and does) still fail, though the failure rate is greatly reduced to 2% using this feedback mechanism.

Carreras and Màrquez [37] built a perceptron-based phrase recognition system. The system is a two-stage pipeline, where the first stage operates at the word level, to filter words and form phrase candidates, and the second stage at the phrase level, to rank phrases and select the optimal ones. Perceptrons are trained for each stage. This paper presented a novel approach for training, which was to define a "global feedback rule" to allow the two perceptrons to be trained together rather than separately, as is typically done. Thus this paper provides an interesting cross between the typical separately-trained models of a feed-forward pipeline and the jointly-trained models in a non-pipelined system. Carreras et al. [38] expanded upon this method to train two stages of a pipeline at once, in order to optimize a global pipeline-final objective rather than the local accuracy objective at each stage. Their method permits errors in the first stage only so long as the errors do not cause errors in the second stage.

Chang et al. [39, 40] present a dependency parsing pipeline model where each "stage" in the pipeline is a single action taken on a pair of words from the input sentence; one of the possible actions is to *step back* to the previously-considered pair, thus allowing for feedback in the pipeline.

3.8.2 Iterative Pipelines

Our work on *pipeline iteration* (Chapter 4) is an obvious example of iterative pipelines. In this work we use constraints derived from the output of later stages in a pipeline to focus the search in earlier stages during a second iteration of the pipeline. Our results showed that such a strategy can improve the accuracy of a pipeline.

Self-trained systems, where output from a system is used to train the same system again, are iteratively-*trained* pipelines. McClosky et al. [152, 153] successfully deployed self-training on the Charniak and Johnson parsing pipeline. They trained the Charniak parser on the Penn Wall Street Journal (WSJ) Treebank [150], used this WSJ-trained model to parse data from the North American News Corpus (NANC), fed the resulting candidates to the Johnson reranker, then used the top candidates output by the reranker to re-train the Charniak parser in a second pass through the pipeline. This pipeline system is noteworthy in several aspects. First, it is one of few examples of pipeline iteration. Second, the iteration only occurs during training; at test-time, an input sentence would be parsed by the self-trained parser (from the second pass) and the output from the parser re-ranked with the top parse output at the end of the pipeline. Finally, the second iteration does not proceed through the entire pipeline; McClosky et al. found that re-using the reranker trained during the initial pass resulted in better performance than re-training the reranker during the second pass. These inconsistencies between training and testing are a puzzle; it seems odd that the parser performed better after re-training but the reranker did not, particularly since matching training to testing conditions is a standard strategy in machine learning. In this pipeline, the testing conditions (output from the self-trained parser) do not match the reranker training conditions (output from the WSJ-trained parser). Chapter 5 will analyze these puzzling results, which also motivate some of the work in Chapters 7 and 8.

The competitive linking model for word-alignment [155] could be argued to be an iterative model, though the iteration occurs within a single stage. Similarly, one might consider minimum error-rate training [165] to be an instance of pipeline iteration. The transformation-based part-of-speech tagger from Brill [27, 28] is, by its nature, an iterative system. Clark et al. [55] is also an iterative system, using the top-ranked candidate output by the final stages of the pipeline to constrain the parse chart in earlier stages of the pipeline.

3.9 Interaction between Pipeline Classes

As has already been mentioned in several of the preceding sections, a number of pipeline classes are tightly tied to each other. For example, the voting and recombination pipelines are often also multi-source pipelines. Similarly, feedback and iterative pipelines can also be considered as multi-source pipelines. As seen in Sections 3.5 and 3.6, every ranking, selection (reranking), and combination (voting) pipeline is also by definition a complete-solution constrained pipelines. Likewise, each extension pipeline is also a partial-solution constrained pipeline. Other classes, while not tightly connected, are also not mutually exclusive. For example, multi-source pipelines can be implemented as either partial-solution or complete-solution constrained pipelines. While it might seem that some of these classes are redundant and therefore unnecessary, we would argue that by including the more abstract classes such as the partial-solution and complete-solution constrained classes, we can capture characteristics that propagate down to each of the subclasses without having to repeatedly verify such characteristics for each subclass.

3.10 Implemented Pipelines

In this section we will describe in detail the finite-state tagging pipeline and context-free parsing pipeline that will be used extensively for the empirical trials to be presented throughout the remainder of this dissertation. The system descriptions will include the classification of these systems within our pipeline framework. In Sections 3.10.3 and 3.10.4, we also describe two other pipelines that we implemented, outside the scope of this dissertation, which provide novel perspectives of the pipeline architecture.

3.10.1 CSLUt Finite-State Tagger

We implemented a finite-stage tagger [109] which we call the Center for Spoken Language Understanding tagger (CSLUt). This tagger can perform any tagging task, including word segmentation, part-of-speech (POS) tagging [110, 85, 183, 187, 160], constituent edge classification [181, 182], shallow parsing or chunking [109, 110], morpheme analysis [162], and named entity recognition. CSLUt can also be trained as a two-stage tagger, where the first set of tags (such as POS tags) are applied at the first stage, then a second set of tags (such as shallow parsing tags) are applied at the second stage. The two layers of tagging can also be trained as a single joint model, for comparison to the pipeline model. Figure 3.14 shows the CSLUt pipeline.

The averaged perceptron algorithm (Section 3.1.5), as presented by Collins [65], is used to train the tagger. During training, the decoding process is performed using a Viterbi search with a second-order Markov assumption (Section 3.2.2). At test-time, the n-best tag sequences are output using either the Viterbi score, or the posterior scores calculated using the forward-backward algorithm (Section 3.2.3), for each tag.

The feature set used in the tagger includes the n-grams of surrounding words and the tags of the preceding words. The n-gram features are represented by the words within a three-word window of the current word. The tag features are represented as unigram, bigram, and trigram tags (i.e., tags from the current and two previous words). These features are based on the feature set implemented by Sha and Pereira [194] for their shallow parser. Additional orthographical features are used for unknown and rare words (words that occur fewer than 5 times in the training data), such as the prefixes and suffixes of the word (up to the first and last four characters of the word), the presence of a hyphen, a digit, or a capitalized letter, following the features implemented by Ratnaparkhi [175]. Table 3.1 summarizes the features implemented in our tagger for POS-tagging

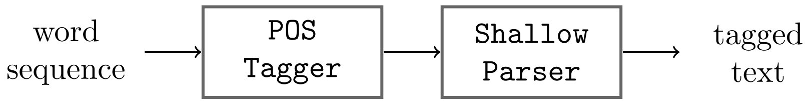


Figure 3.14: The CSLU tagging pipeline, used for empirical parsing experiments throughout this dissertation.

and shallow parsing. In this table the ρ features are instantiated as POS-tags and τ features are instantiated as shallow chunk tags. Note that the orthographic feature templates, including the prefix (e.g., $w_i[0..1]$) and suffix (e.g., $w_i[n\text{-}2..n]$) templates, are only activated for unknown and rare words.

We follow Ratnaparkhi [173] in restricting our tagger's search space for each lexical item. Any word which occurs more than five times in the training data may only be tagged with tags that occurred with that word, which Ratnaparkhi [173] refers to as a "Tag Dictionary;" the tags for words which occur fewer than five times are unrestricted. We implemented this restriction for both POS-tagging and shallow parsing, though in shallow parsing the restriction is based on POS tags rather than lexical items; words could only be tagged with those shallow chunk tags that have occurred with that word's POS tag. We chose to rely on POS tags in order to improve the generality of the parser, though of course the restriction could be moved to the word itself which would decrease the search space and thus increase the efficiency of the search. This improvement doubles the efficiency of the tagger with no loss in accuracy; in fact, the accuracy tends to increase slightly, which is consistent with the results shown by Ratnaparkhi [173]. We also only extracted the orthographic features for the rare or unseen words.

CSLUt was designed such that the set of classes over which it searches is defined based on the observed tag classes in its training set. Thus it is a simple matter to train CSLUt as either a POS-tagger, an NP-chunker [172], or a shallow parser [207]. In [109] we showed that CSLUt is a competitive system in both NP-chunking and the CoNLL-2000 Chunking task. Alternately, as mentioned above, CSLUt could also be trained as a joint POS-tagging and shallow parsing model.

Our CSLUt tagging pipeline can be run as an unconstrained *creation* model, or take constraints (in a number of different formats) as input. Thus the first stage of our tagging pipeline can be either a *creation*, *combination*, or *selection* stage. The second stage of the CSLUt pipeline shown in Figure 3.14 (p. 56) is an *extension* stage, though it can also function as a *combination* or *selection* stage. The tagging pipeline has only been implemented as a *feed-forward* pipeline, although it has the capacity to function as an *iterative* pipeline.

We discuss experiments using this tagger extensively in Chapters 4 and 5.

3.10.2 Charniak & Johnson Context-Free Parser & MaxEnt Reranker

Throughout this dissertation we will make extensive use of the Charniak and Johnson parsing pipeline [44].[16] This pipeline contains three stages as shown in Figure 3.15; the first two stages, delineated by a black box in the figure, are the parsing stages (based on the Charniak parser [41]), and the last stage is a reranker.

The parser utilizes a "coarse-to-fine" model, so the second parsing stage is more complex than the first. In the parser, an input sentence is coarsely parsed based on a "vanilla" probabilistic context-free grammar (PCFG), which outputs a parse chart to the second stage. The second stage then prunes the entries in the chart to find the coarse-grained states corresponding to high-probability fine-grained states, and the unpruned entries are evaluated based on a more-complex PCFG with head-percolation and parent- and grandparent-annotation included in the grammar.

[16] We gratefully acknowledge Eugene Charniak and Mark Johnson for their help with the parser and reranker documented in their paper, and for making their code available (at ftp://ftp.cs.brown.edu/pub/nlparser/).

Φ for POS Tagging		
ρ_i		
ρ_{i-1}, ρ_i		
ρ_{i-2}, ρ_i		
$\rho_{i-2}, \rho_{i-1}, \rho_i$		
ρ_i or ρ_i, ρ_{i-1}	w_i	
	w_{i-1}	
	w_{i+1}	
	w_{i-2}	
	w_{i+2}	
	w_{i-1}, w_i	
	w_i, w_{i+1}	
	$w_i[0]$	$\diamond$
	$w_i[0..1]$	$\diamond$
	$w_i[0..2]$	$\diamond$
	$w_i[0..3]$	$\diamond$
	$w_i[n]$	$\diamond$
	$w_i[n\text{-}1..n]$	$\diamond$
	$w_i[n\text{-}2..n]$	$\diamond$
	$w_i[n\text{-}3..n]$	$\diamond$
	$w_i \subseteq$ Digit	$\diamond$
	$w_i \subseteq$ UpperCase	$\diamond$
	$w_i \subseteq$ Hyphen	$\diamond$

Φ for Shallow Parsing		
τ_i		
τ_{i-1}, τ_i		
τ_{i-2}, τ_i		
$\tau_{i-2}, \tau_{i-1}, \tau_i$		
τ_i or τ_i, τ_{i-1}	w_i	
	w_{i-1}	
	w_{i+1}	
	w_{i-2}	
	w_{i+2}	
	w_{i-1}, w_i	
	w_i, w_{i+1}	
	$w_i[0]$	$\diamond$
	$w_i[0..1]$	$\diamond$
	$w_i[0..2]$	$\diamond$
	$w_i[0..3]$	$\diamond$
	$w_i[n]$	$\diamond$
	$w_i[n\text{-}1..n]$	$\diamond$
	$w_i[n\text{-}2..n]$	$\diamond$
	$w_i[n\text{-}3..n]$	$\diamond$
	$w_i \subseteq$ Digit	$\diamond$
	$w_i \subseteq$ UpperCase	$\diamond$
	$w_i \subseteq$ Hyphen	$\diamond$
	ρ_i	
	ρ_{i-1}	
	ρ_{i-1}, ρ_i	
	ρ_{i+1}	
	ρ_i, ρ_{i+1}	
	$\rho_{i-1}, \rho_i, \rho_{i+1}$	
	ρ_{i-2}	
	ρ_{i-2}, ρ_{i-1}	
	$\rho_{i-2}, \rho_{i-1}, \rho_i$	
	ρ_{i+2}	
	ρ_{i+1}, ρ_{i+2}	
	$\rho_i, \rho_{i+1}, \rho_{i+2}$	

$\diamond$ *Orthographic Features*

Table 3.1: Feature templates defined for the CSLUt finite-state tagger.

These first two stages can function as a stand-alone parsing system. The third stage of this pipeline is the reranker, which increases the accuracy of the output parse but also adds complexity to the training and testing of the pipeline. The reranker trains on and takes as input an n-best list of parse candidates from the parser, and reranks (or re-scores) each candidate set according to a detailed feature set which uses larger context than the context-free parser. Note that the reranker optimized the conditional likelihood of the "oracle" solution in the input n-best lists. See Charniak and Johnson [44] for more details.

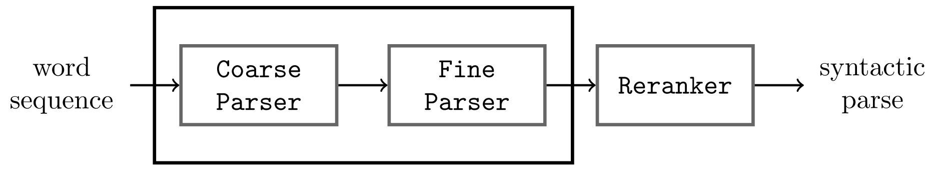


Figure 3.15: The Charniak and Johnson [44] parsing pipeline, used for empirical parsing experiments throughout this dissertation.

We made substantial modifications to this code for the controlled experimentation presented in this dissertation. We experimented with different feature sets in the reranker stage, including features based on shallow parses extracted from each full context-free parse, but ultimately settled on the features described in [44]. We also experimented with maximum-margin optimization but saw very little empirical difference using this optimization method. Finally, in order to train the reranker on very large n-best lists (1000+) and their correspondingly large feature sets, we modified the feature representation to consume far less memory than required by the out-of-the-box vector representation.[17]

The Charniak and Johnson [44] parsing pipeline includes a *creation* stage, a *combination* stage, and a *selection* stage. We modified the parsing stages in this pipeline to take a number of different types of *partial-solution* constraints as input. These modifications allowed us to implement *iteration* in this pipeline (see Chapter 4). The final stage of the Charniak/Johnson pipeline is the reranker, which we have noted already is a *selection* stage rather than a *ranking* stage, since the model is evaluated only on the accuracy of its one-best output. We modified the reranking stage to output the entire set of input candidates, ranked according to the score calculated by the reranker model, rather than just the top-ranked candidate. This modification will allow us to measure characteristics of the constrained space as a whole, as well as the reranker model's distribution over the space (see Chapter 8). We also experimented with large-margin optimization in the reranker, i.e., training the model as a *ranking* model.

We discuss experiments using this parsing pipeline extensively throughout the remainder of this dissertation.

3.10.3 Chart Cell Closure Pipeline

There have been several parsing pipelines that make use of finite-state chunkers early in the pipeline to constrain downstream context-free parsers; imposing such constraints has been shown to have an efficiency [95] and/or an accuracy [110] benefit. Glaysher and Moldovan [95] demonstrated an efficiency gain by explicitly disallowing entries in chart cells that would result in constituents that cross chunk boundaries. We [110] demonstrated that high precision constraints on early stages of the Charniak and Johnson [44] pipeline—in the form of base phrase constraints derived either from a chunker or from later stages of an earlier iteration of the same pipeline—achieved significant accuracy improvements, by moving the pipeline search away from unlikely areas of the search space. Both of these approaches (and others, including the well-known Ratnaparkhi [175] parser) achieve their improvements by *ruling out* parts of parse chart for downstream context-free parsers, and the gain can either be realized in efficiency (same accuracy, less time) or accuracy (same time, greater accuracy). Parts of the parse chart are ruled out just when they are inconsistent with the output of the chunker: the constraints are a *by-product* of chunking.

In [181], we build classifiers that *more directly* address the problem of "closing" chart cells

[17] Our modifications will be made available online.

to entries, rather than extracting this information from taggers or chunkers built for a different purpose. We build two classifiers, which tag each word in the sequence with a binary class label. The first classifier decides if the word can begin a constituent of span greater than one word; the second classifier decides if the word can end a constituent of span greater than 1. Given a chart cell (i, j) with start word w_i and end word w_j, where $j > i$, that cell can be "closed" to entries if the first classifier decides that w_i cannot be the first word of a multi-word constituent or if the second classifier decides that w_j cannot be the last word in a multi-word constituent. In such a way, we optimize classifiers specifically for the task of constraining chart parsers.

We also presented a method for "closing" a sufficient number of chart cells to ensure *quadratic* worst-case complexity of context-free inference. By applying such constraints to the state-of-the-art Charniak [41] parsing pipeline, which resulted in no accuracy loss when the constraints were applied, we provided empirical evidence to show that this $O(n^2)$ bound can be achieved without impacting parsing accuracy. Thus we derived and applied finite-state constraints so as to *guarantee* a reduction in the worst-case complexity of the context-free parsing pipeline from $O(n^3)$ in the length of the string n to $O(n^2)$ by closing chart cells to entries. In [182] we extended these cell-closure methods to an exact inference CYK parser, and proved that a different method of imposing constraints on words beginning or ending multi-word constituents can give $O(n\log^2 n)$ or $O(n)$ worst-case complexity.

Our cell-closure pipeline system introduced the concept of training upstream models to directly constrain the search space of a downstream model. By taking into account some information about the downstream stage—namely, that it represented its search space as a parse chart—we were able to achieve both accuracy and efficiency gains by placing a guaranteed limit on the complexity of the downstream search algorithm.

3.10.4 Morphological Mimic Pipeline

In [162] we investigated simulating rule-based and black-box NLP systems with stochastic models, by creating pipelines with the non-stochastic (or black-box) systems upstream and the stochastic models downstream. The benefit of such a *mimic* pipeline is in using the stochastic models to output numeric confidence estimates for the rule-based and black-box NLP systems. Numeric confidence estimates enable both minimum Bayes risk–style optimization as well as principled system combination. Minimum Bayes risk inference enables the tuning of NLP systems to tune between high precision and high recall output. System combination can unite the complementary strengths of independent systems. Unfortunately, the NLP systems that we would like to optimize or combine do not always produce weights from which confidence estimates may be calculated. In some domains, knowledge-based systems are widely used and are effective; for example, the best stemming, tokenization, and morphological analyzers for many languages are hard clustering approaches that do not involve weights or even yield alternative analyses. For other tasks, weights may be used system-internally, but are not immediately accessible to the end-user; such a system is a black-box to the user.

In our specific experiments, we simulated ParaMor [161], a rule-based system for unsupervised morphology induction, with our CSLUt statistical tagger [109, 110]. We trained our CSLUt tagger to identify—based on output from ParaMor—for each character in a given word, whether or not there is a morpheme boundary at that character, and for each morpheme boundary, whether that morpheme is a stem or a suffix. The feature set used in the tagger included just the character n-grams up to three characters on either side of the current character[18] and the unigram, bigram, and trigram morpheme-tags (i.e., tags from the current and two previous characters). Thus CSLUt was

[18] Thus in a word like "quickly", the character-features for tagging the letter 'c' would be: 'quic', 'uic', 'ic', 'c', 'ck', 'ckl', and 'ckly'.

trained directly on output from ParaMor, creating a mimic system. The tagger outputs posterior probabilities, which can serve as confidence measures for the original systems. Leveraging these newfound confidence scores, we pursued minimum Bayes risk–style thresholding of tags (for higher morpheme recall) as well as principled system combination approaches (for higher overall accuracy), resulting in improved morpheme identification on a Hungarian corpus by 5.9% absolute F-score.

Our novel concept of mimic pipelines is not, by any means, limited to morphological analysis, and could prove beneficial in many other areas of NLP as well as other research fields.

3.11 Summary

The goal of this chapter was to conduct a large-scale classification of existing pipeline systems, situating each system within the pipeline framework defined by this dissertation. Such a large-scale classification was intended to demonstrate that the framework is sufficient to cover any existing pipeline system, and to provide instruction for the classification of any future pipeline implementations.

Recall that the reasoning behind the pipeline framework for classifying pipeline systems was that pipelines in the same classes would behave similarly under varied conditions. Thus in the remainder of the dissertation, we will systematically vary pipeline conditions by altering characteristics of the pipeline constraints, including the accuracy, diversity, regularity, and peakedness of the constraints. We will also use different classes of pipelines for our empirical trials. By comparing the results of using these constraints in various parsing pipelines for various tasks, this research will provide insight as to the influence of constraints on pipeline performance.

Part II

Pipeline Techniques

One of the benefits of formalizing a framework for pipeline systems is the ability to draw on this framework to create generalized techniques for pipeline improvement. The next two chapters of this dissertation present techniques for improving a pipeline by using different message-passing strategies (Chapter 4), and by changing how models in the pipeline interact with other models (Chapter 5). While the empirical results are presented for parsing pipelines, we will emphasize the generalizable nature of these techniques.

Chapter 4

Pipeline Iteration

This chapter presents *pipeline iteration*,[1] a general pipeline technique inspired by the iterative class of pipelines presented in Chapter 2. The basic idea of pipeline iteration is to use constraints derived from the output of later stages in a pipeline to focus the search in earlier stages during subsequent iterations of the pipeline. Results will show that such a strategy can be used to improve the accuracy of a pipeline.

We investigate pipeline iteration within the context of the Charniak and Johnson [44] parsing pipeline, by constraining parses to be consistent with a set of *base phrases* (which will be defined in Section 4.2). The Charniak [41] parsing pipeline has been extensively studied over the past decade, with a number of papers focused on improving early stages of the pipeline [42, 34, 26, 102, 45] as well as many focused on optimizing final parse accuracy [41, 44, 153]. This focus on optimization has made system improvements very difficult to achieve, yet our relatively simple technique yields statistically significant improvements, making pipeline iteration a promising approach for other tasks.

4.1 Iterated Constraints

It may seem surprising that later stages, already constrained to be consistent with the output of earlier stages, can profitably inform these same earlier stages during a second pass through the pipeline. However, the richer models used in later stages of a pipeline provide a better distribution over the subset of possible solutions produced by the early stages, effectively resolving some of the ambiguities that account for much of the original variation. If an earlier stage is then constrained in a second pass not to vary with respect to these resolved ambiguities, it will be forced to find other variations, which may include better solutions than were originally provided.

To give a rough illustration, consider the Venn diagrams in Figure 4.1. In Figure 4.1(i), set A represents the original subset of possible solutions passed along by the earlier stage, and the dark shaded region represents high-probability solutions according to later stages. If some constraints are then extracted from these high-probability solutions, defining a subset of solutions (S) that rule out some of A, the early stage will be forced to produce a different set (B). Constraints derived from later stages of the pipeline focus the search in an area believed to contain high-quality candidates.

Another scenario is to use a different model altogether to constrain the pipeline. In this scenario, represented in Figure 4.1(ii), the other model constrains the early stage to be consistent with some subset of solutions (S), which may be largely or completely disjoint from the original set A. Again, a different set (B) results, which may include better results than A. Whereas when iterating we

[1] Much of the work in this chapter was published in ACL'07 [110].

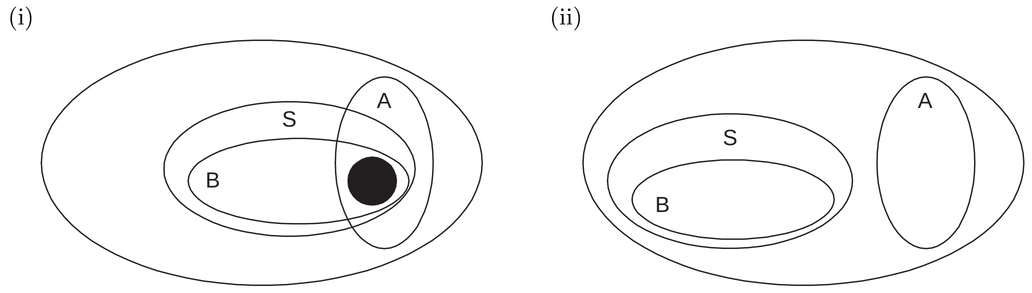


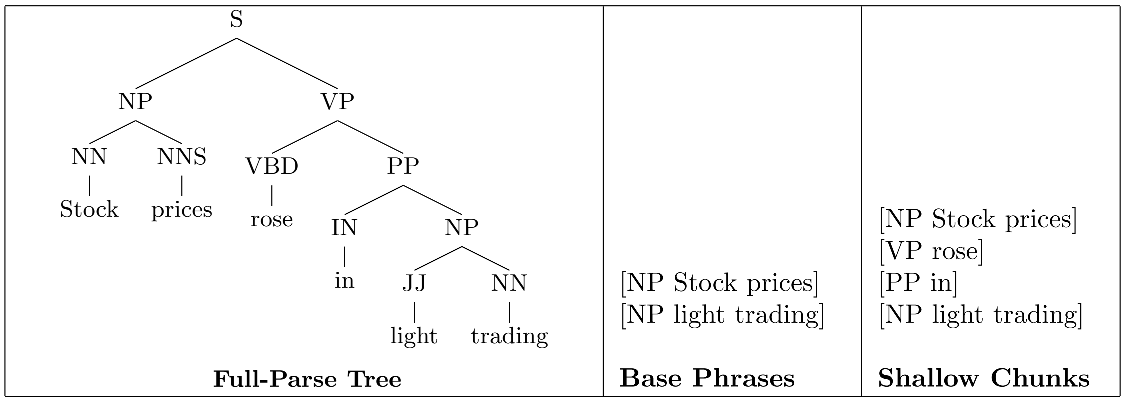
Figure 4.1: Two Venn diagrams, representing (i) constraints derived from later stages of an iterated pipelined system; and (ii) constraints derived from a different model.

are guaranteed that the new subset S will overlap at least partially with the original subset A, that is not the case when making use of constraints from a separate model.

4.2 Base Phrases

In this section we will define *base phrases*. Following Ratnaparkhi [175], we define a base phrase as any parse node with only preterminal children (review Section 3.3.2 for terminology definitions). Unlike the "shallow chunks" defined for the CoNLL-2000 Shared Task [207], base phrases correspond directly to constituents that appear in full (context-free) parses, and hence can provide a straightforward constraint on edges within a chart parser. In contrast, shallow chunks collapse certain non-constituents—such as auxiliary chains—into a single phrase, and therefore are not directly applicable as constraints on a chart parser. Figure 4.2 compares the set of base phrases (center) versus the set of shallow chunks (right) which would be extracted from the full parse tree (left).

We have two systems capable of producing base-phrase annotations for a string. One is our CSLUt shallow parser (see Section 3.10.1 for a full system description), which we trained on base phrases extracted from the Penn Wall St. Journal (WSJ) Treebank [150]. The treebank trees are pre-processed identically to the procedure for training the Charniak parser, e.g., empty nodes and function tags are removed. The second is the Charniak and Johnson [44] parsing pipeline (see Section 3.10.2 for a full system description); we extracted base phrases from the full-parse



Base Phrases	Shallow Chunks
	[NP Stock prices]
	[VP rose]
[NP Stock prices]	[PP in]
[NP light trading]	[NP light trading]

Figure 4.2: The base phrases (center) and shallow chunks (right) as extracted from a full-parse tree (left). Note that neither the S, the VP, nor the PP nodes in the full parse qualify as base phrases because they each have at least one non-preterminal child.

Parser		**Base Phrases**	**Shallow Chunks**
Charniak and Johnson [44]	parser-best	91.9	94.4
	reranker-best	92.8	94.8
CSLUt shallow parser		91.7	94.3

Table 4.1: F-scores on WSJ section 24 of output from two parsers on the similar tasks of base-phrase parsing and shallow chunking. For evaluation of the Charniak/Johnson full parser, base phrases and shallow chunks are deterministically extracted from the full-parse output via a simple extraction script.

tree output, via a simple script to extract nodes with only preterminal children. Note that the second-stage parser and the final-stage reranker both output full-parse trees from which we can extract base phrases.

Table 4.1 shows the bracketing accuracy of the CSLUt shallow parser and the Charniak/Johnson full parser on both the base phrase and shallow chunking tasks for WSJ section 24; each system was trained on WSJ sections 02-21. In the table we report on the *parser-best* and *reranker-best* parses; parser-best is an intrinsic, mid-pipeline evaluation of the parse ranked highest by the parsing model, and reranker-best is an extrinsic pipeline-final evaluation of the parse ranked highest by the reranking model. From this table we can see that base phrases are substantially more difficult than shallow chunks to annotate. Output from the CSLUt shallow parser is roughly as accurate as output extracted from the Charniak parser-best trees, though a fair amount below output extracted from the reranker-best trees.

In addition to using base phrase constraints from these two sources independently, we also looked at *combining* the predictions of both to obtain more reliable constraints. Section 4.4 presents two methods of combining output from multiple parsers.

4.2.1 To Constrain a Parser

In order to constrain a parser with base phrases, as defined in the previous section, we require full parses to be *consistent with* the base-phrase tree provided as input to the parser, i.e., any valid parse must contain all of the base-phrase constituents in the constraining tree. The full-parse tree in Figure 4.3(b), for example, is consistent with the base-phrase tree in Figure 4.3(a). Implementing these constraints in a parser is straightforward, one of the advantages of using base phrases as constraints. Since the internal structure of base phrases is, by definition, limited to preterminal children, we can constrain the entire parse by constraining the parents of the

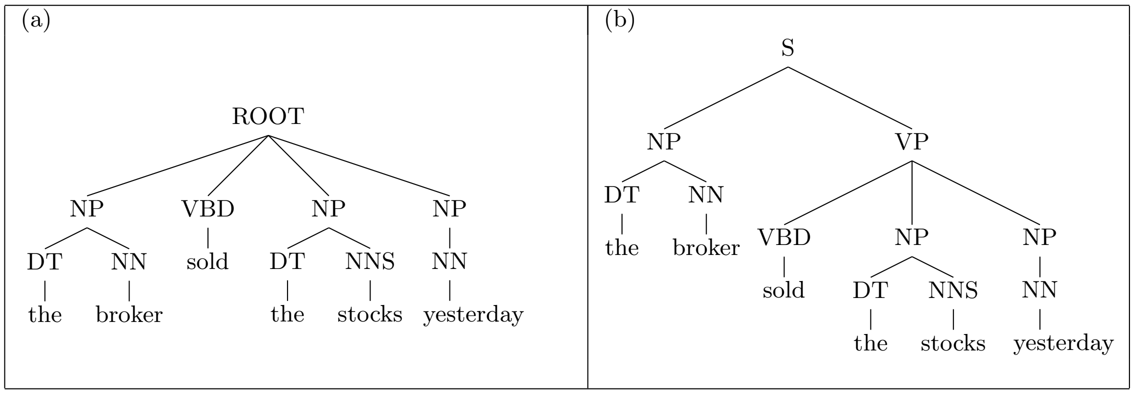


Figure 4.3: Base-phrase tree (a) as produced to constrain a parser and full-parse tree (b) consistent with the constraining base-phrase tree (a).

appropriate preterminal nodes. For any preterminal that occurs within the span of a constraining base phrase, the only valid parent is a node matching both the span (start and end points) and the label of the provided base phrase. All other parent-nodes proposed by the parser are rejected. In such a way, for any parse to cover the entire string, it would have to be consistent with the constraining base-phrase tree.

Words that fall outside of any base-phrase constraint are unconstrained in how they attach within the parse. Thus a constrained full-parse tree might contain more base phrases than were in the constraint set; i.e., the base phrases in the constrained output may actually be a superset of the base-phrase constraint set. Note also that a smaller set of constraints will result in a larger search space downstream, so a base-phrase tree with few words covered by base phrases will be less constraining than a base-phrase tree with many words covered by base phrases.

4.3 Base-Phrase–Constrained Results

For the experiments reported in this chapter we use the Charniak and Johnson [44] state-of-the-art parsing pipeline, which we summarize briefly here but described in-depth in Section 3.10.2. The well-known coarse-to-fine parser [41] is a two-stage parsing pipeline, in which the first stage uses a vanilla PCFG to populate a chart of parse constituents. The second stage, constrained to only those items in the first-stage chart, uses a refined grammar to generate an n-best list of parse candidates. Charniak and Johnson [44] extended this pipeline with a discriminative MaxEnt model to rerank the n-best parse candidates, deriving a significant benefit from the richer model employed by the reranker. Following Charniak and Johnson [44], we set the parser to output 50-best parses for all experiments described below.

Unless stated otherwise, all reported results will be F-scores on WSJ section 24 of the Penn WSJ Treebank, which was our development set. Training data was WSJ sections 02-21, with section 00 as heldout data. Crossfold validation (20-fold with 2,000 sentences per fold) was used to train the reranker for every condition. Evaluation was performed using `evalb` under standard parameterizations (see Section 3.3.2). WSJ section 23 was used only for final testing and statistical significance is only reported for that section.

For our experiments, we modified the Charniak/Johnson parser to allow us to optionally provide base-phrase trees to constrain the first stage of parsing: during chart construction, we disallow any constituents that conflict with the constraints, as described in detail in Section 4.2.1. In our approach, we constrained the parser with base phrases but put no restrictions on the preterminal labels, even within the base phrases. We also normalized for punctuation. In the rare cases where the parser failed to find a valid parse with the constraints, we lifted the constraints and allowed any parse constituent originally proposed by the first stage of the parser.

Our experiments will demonstrate the effects of constraining the parser under several different conditions. First, we present results without any constraints on the parser, as our baseline condition. Next, we consider two possible sources of base phrase constraints: (1) the base phrases extracted from the full-parse output of the reranker; and (2) the base phrases output by the CSLUt shallow parser.

For each of our experimental conditions we constructed a simple parsing pipeline, as shown in Figure 4.4. Note that at the core of each of our pipelines is the Charniak and Johnson [44] coarse-to-fine parser and MaxEnt reranker (the shaded stages in the pipelines in Figure 4.4), with the reranker always as the pipeline-final stage.

Unconstrained

For our baseline system, we run the Charniak and Johnson [44] parser and reranker with default parameters. Treebank-tokenized text is input to the parser and, as mentioned previously, 50-best

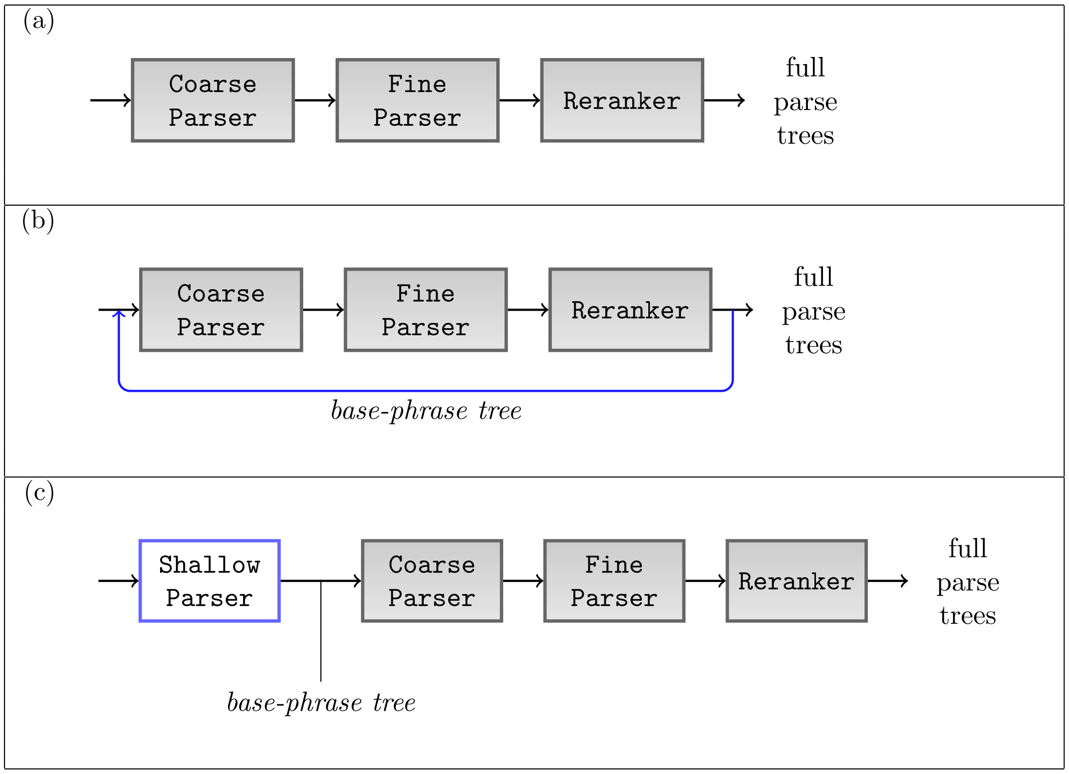


Figure 4.4: Experimental conditions to test the pipeline iteration technique: (a) baseline, unconstrained pipeline; (b) classic iterated pipeline; and (c) constrained, non-iterated pipeline.

parse candidates are outputs to the reranker.

Reranker-constrained

We use the reranker-constrained condition to examine the effects of pipeline iteration, with no input from other models outside the pipeline. Taking the reranker-best full parse as output under the condition of unconstrained search, we extract the corresponding base-phrase tree and make a second pass through the parsing pipeline, now with base-phrase constraints from the reranker. The pipeline for this experimental condition is shown in Figure 4.4(b).

CSLUt-Constrained

The CSLUt-constrained condition provides a comparison point of *non-iterated* constraints. Under this condition, the one-best base-phrase tree output by the CSLUt shallow parser is input as a constraint to the Charniak parser. We run the parser and reranker as before, now constrained by output from the shallow parser, as shown in Figure 4.4(c). Note that this condition is *not* an instance of pipeline iteration; however, we include this condition to conclusively demonstrate that any performance improvements are not due simply to the use of base-phrase constraints, but rather from using pipeline iteration to derive the constraints.

The effects of constraining the parser under these various conditions are shown in Table 4.2. We evaluated the one-best parse candidates before and after reranking (*parser best* and *reranker best*, respectively), as well as evaluating for the best-possible F-score in the n-best list (*oracle best*). The labeled recall, precision, and F-score accuracy of each set of base-phrase constraints are shown in the last three columns of the table.

Constraints	Parser Best	Reranker Best	Oracle Best	Base-Phrase R	Base-Phrase P	Base-Phrase F
Baseline (Unconstrained)	88.9	90.2	**96.0**	–	–	–
Reranker-constrained	89.6	90.5	95.1	92.2	93.3	92.8
CSLUt-constrained	88.4	89.5	94.1	91.3	92.0	91.7

Table 4.2: Full-parse F-scores on WSJ section 24. The unconstrained search (first row) provides a baseline comparison for the effects of constraining the search space. The bottom two rows demonstrate the effect of two different sources of constraints; the last three columns show the recall (R), precision (P), and F-score (F) of the constraints produced for each condition.

Constraining the parser to the base phrases produced by the reranker provides a 0.7 percent improvement in the parser-best accuracy, and a 0.3 percent improvement after reranking. Constraining the parser to the base phrases produced by the CSLUt shallow parser (CSLUt-constrained) hurts performance by half a point.

The oracle rate decreases under all of the constrained conditions as compared to the baseline, demonstrating that the parser was prevented from finding some of the best solutions that were originally found. However, the improvement in parser-best and reranker-best F-score, under all conditions *except* the non-iterated CSLUt constraints condition, shows that the iterated constraints assisted the parser in achieving high-quality solutions despite this degraded oracle accuracy of the lists. Furthermore, the fact that the parser-best F-score increased under all iterated conditions indicates that we have resolved either search or model errors (defined on p. 22); results later in the chapter (Section 4.5.2) will help us to tease apart the differences in the types of errors resolved by the pipeline iteration technique.

4.4 Combining Parser n-best Lists

Let us now take a brief side-trip to examine two different methods of combining parser outputs; we will return to results in Section 4.5. The first method is to simply take the union of two n-best lists. The second is more complex, recombining the elements of each tree in the n-best lists to create new parses that may not have existed in the original n-best lists.

4.4.1 Union

We take the union of two n-best lists of parse candidates as a straightforward set union. The only noteworthy step of the union operation is in calculating the scores of the duplicate candidates, i.e., the parse candidates which were in each of the original lists. Since the score of a candidate has been shown to be a highly-informative feature for a downstream reranker [44, 68], we used Equation 4.1 to mix the scores from each source.

Let $\mathcal{T}$ be the set-union of the two n-best lists. For all trees $T \in \mathcal{T}$, let $\mathrm{P}_1(T)$ be the probability of T in the first n-best list, and $\mathrm{P}_2(T)$ the probability of T in the second n-best list. Then, we define $\mathrm{P}(T)$ as follows:

$$\mathrm{P}(T) \quad = \quad \alpha \frac{\mathrm{P}_1(T)}{\sum\limits_{T' \in \mathcal{T}} \mathrm{P}_1(T')} + \frac{\mathrm{P}_2(T)}{\sum\limits_{T' \in \mathcal{T}} \mathrm{P}_2(T')} \tag{4.1}$$

where the parameter α dictates the relative weight of P_1 versus P_2 in the combination. Since P_1 and P_2 are normalized in this equation, they are not required to be true probabilities. Furthermore, the output of this equation will also not necessarily be a true probability, since α can range from

$[-\infty, \infty]$. Clearly, larger α values place more weight on the scores from the first n-best list, while smaller α values place more weight on the second n-best list.

4.4.2 Recombination

In this section we explore a more complex method of combining sets of parse candidates, which we term *recombination*.

In order to select high-likelihood constraints for the pipeline, we may want to extract annotations with high levels of agreement ("consensus hypotheses") between candidates. In addition, we may want to favor precision over recall to avoid erroneous constraints within the pipeline as much as possible. Here we discuss how a technique presented in Goodman's thesis [100] can be applied to do this.

We will first present this within a general chart parsing approach, then move to how we use it for n-best lists. Let $\mathcal{T}$ be the set of trees for a particular input, and let a parse $T \in \mathcal{T}$ be considered as a set of labeled spans. Then, for all labeled spans $X \in T$, we can calculate the posterior probability $\gamma(X)$ as follows:

$$\gamma(X) \quad = \quad \sum_{T \in \mathcal{T}} \frac{\mathrm{P}(T)[\![X \in T]\!]}{\sum_{T' \in \mathcal{T}} \mathrm{P}(T')} \quad \text{where} \quad [\![X \in T]\!] \quad = \quad \begin{cases} 1 & \text{if } X \in T \\ 0 & \text{otherwise.} \end{cases} \tag{4.2}$$

Note P is normalized in Equation 4.2, such that the posterior probability γ is a true probability.

Goodman [98, 100] presented a method for using the posterior probability of constituents to maximize the expected labeled recall of binary branching trees, as follows:

$$\widehat{T} \quad = \quad \operatorname*{argmax}_{T \in \mathcal{T}} \sum_{X \in T} \gamma(X) \tag{4.3}$$

Essentially, find the tree with the maximum sum of the posterior probabilities of its constituents. This is done by computing the posterior probabilities of constituents in a chart, typically via the Inside-Outside algorithm [10, 138], followed by a final CYK-like pass to find the tree maximizing the sum.

Now we will discuss how to use this approach to recombine n-best output from multiple parsers. For simplicity, we will here discuss the combination of two n-best lists, though it generalizes in the obvious way to an arbitrary number of lists. To perform this combination, we construct a CYK chart of each constituent in each parse from each of the n-best lists. For all labeled spans $X \in T$ in the CYK chart, we can calculate the posterior probability $\gamma(X)$ as in Equation 4.2 then extract from the chart the maximum-scoring tree.

For our experiments in this chapter, we combined two n-best lists of base-phrase trees using Equation 4.2. Even though there is no hierarchical structure in base-phrase annotations, they can be represented as flat trees, as shown in Figure 4.5(a). We constructed a CYK chart from the two lists being combined, using Equation 4.1 to define $\mathrm{P}(T)$ in Equation 4.2.

We wish to consider every possible combination of the base phrases, so for the final CYK-like pass to find the argmax tree, we included rules for attaching each preterminal directly to the root of the tree, in addition to rules permitting any combination of hypothesized base phrases. Thus we allowed any combination of constituents that results in a tree—even one with no internal structure. Consider the trees in Figure 4.5. Figure 4.5(a) is a shallow parse with three NP base phrases; Figure 4.5(b) is the same parse where the ROOT production has been binarized for the final CYK-like pass, which requires binary productions. If we include, in the CYK chart, productions of the form 'ROOT $\rightarrow$ X ROOT' and 'ROOT $\rightarrow$ X Y' for all non-terminals X and Y (including POS tags), then any tree-structured combination of base phrases hypothesized in either

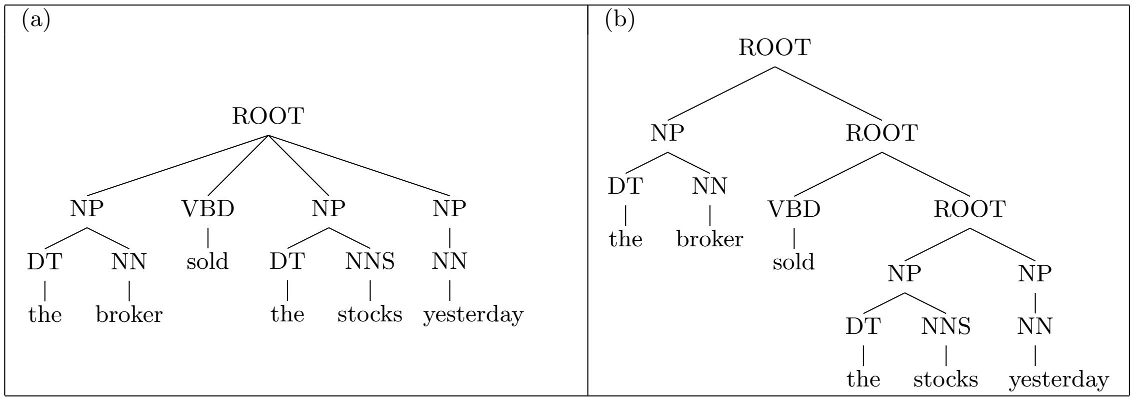


Figure 4.5: Base-phrase trees (a) as produced for an n-best list and (b) after root-binarization for n-best list combination.

n-best list will be allowed, including the one with no base phrases at all.[2]

Precision/Recall Tradeoff Decoding

Recall that Equation 4.3 uses the posterior probability of constituents (Equation 4.2) to maximize the expected labeled *recall* of binary branching trees; in strictly binary branching trees recall and precision are equivalent. For non-binary branching trees, where precision and recall may differ, Goodman [100, Ch. 3] proposed the following combined metric for a *precision/recall tradeoff*:

$$\widehat{T} \;=\; \operatorname*{argmax}_{T\in\mathcal{T}} \sum_{X\in T} (\gamma(X) - \lambda) \tag{4.4}$$

where λ ranges from 0 to 1. Setting λ=0 is equivalent to Equation 4.3 and thus optimizes recall, and setting λ=1 optimizes precision, as shown below.[3]

The following is a formal derivation of the precision/recall tradeoff method. Recall that $\mathcal{T}$ is the set of trees for a particular input, and each $T \in \mathcal{T}$ is considered as a set of labeled spans. If τ is the reference tree, the labeled precision (LP) and labeled recall (LR) of a T relative to τ are defined as

$$\text{LP} \;=\; \frac{|T\cap\tau|}{|T|} \qquad\qquad \text{LR} \;=\; \frac{|T\cap\tau|}{|\tau|} \tag{4.5}$$

where $|T|$ denotes the size of the set T.

A metric very close to labeled recall (LR) is $|T\cap\tau|$, the number of nodes in common between the tree and the reference tree. To maximize the expected value ($\mathcal{E}$) of this metric, we want to find the tree $\widehat{T}$ as follows:

$$\begin{aligned}
\widehat{T} &= \operatorname*{argmax}_{T\in\mathcal{T}} \mathcal{E}\left[|T \bigcap \tau|\right] \\
&= \operatorname*{argmax}_{T\in\mathcal{T}} \sum_{T'\in\mathcal{T}} \frac{\mathrm{P}(T')\,[|T\bigcap T'|]}{\sum_{T''\in\mathcal{T}} \mathrm{P}(T'')} \\
&= \operatorname*{argmax}_{T\in\mathcal{T}} \sum_{T'\in\mathcal{T}} \frac{\mathrm{P}(T')\sum_{X\in T}[\![X\in T']\!]}{\sum_{T''\in\mathcal{T}} \mathrm{P}(T'')}
\end{aligned}$$

[2] For the purpose of finding the argmax tree in Equation 4.4, we only sum the posterior probabilities of base-phrase constituents, and not the ROOT symbol or POS tags.

[3] Note that our notation differs slightly from that in [100], though the approaches are formally equivalent.

$$
\begin{aligned}
&= \operatorname*{argmax}_{T\in\mathcal{T}} \sum_{X\in T}\sum_{T'\in\mathcal{T}} \frac{\mathrm{P}(T')[\![X\in T']\!]}{\sum_{T''\in\mathcal{T}}\mathrm{P}(T'')} \\
&= \operatorname*{argmax}_{T\in\mathcal{T}} \sum_{X\in T}\gamma(X) \qquad (4.6)
\end{aligned}
$$

This exactly maximizes the expected labeled recall in the case of binary branching trees, and is closely related to labeled recall for non-binary branching trees. Similar to maximizing the expected number of matching nodes, we can minimize the expected number of non-matching nodes, for a metric related to labeled precision (LP):

$$
\begin{aligned}
\widehat{T} &= \operatorname*{argmin}_{T\in\mathcal{T}} \mathcal{E}\left[|T| - |T\bigcap\tau|\right] \\
&= \operatorname*{argmax}_{T\in\mathcal{T}} \mathcal{E}\left[|T\bigcap\tau| - |T|\right] \\
&= \operatorname*{argmax}_{T\in\mathcal{T}} \sum_{T'\in\mathcal{T}} \frac{\mathrm{P}(T')\left[|T\bigcap T'| - |T|\right]}{\sum_{T''\in\mathcal{T}}\mathrm{P}(T'')} \\
&= \operatorname*{argmax}_{T\in\mathcal{T}} \sum_{T'\in\mathcal{T}} \frac{\mathrm{P}(T')\sum_{X\in T}([\![X\in T']\!]-1)}{\sum_{T''\in\mathcal{T}}\mathrm{P}(T'')} \\
&= \operatorname*{argmax}_{T\in\mathcal{T}} \sum_{X\in T}\sum_{T'\in\mathcal{T}} \frac{\mathrm{P}(T')([\![X\in T']\!]-1)}{\sum_{T''\in\mathcal{T}}\mathrm{P}(T'')} \\
&= \operatorname*{argmax}_{T\in\mathcal{T}} \sum_{X\in T}(\gamma(X)-1) \qquad (4.7)
\end{aligned}
$$

Finally, we can combine these two metrics in a linear combination. Let λ be a mixing factor from 0 to 1. Then we can optimize the weighted sum:

$$
\begin{aligned}
\widehat{T} &= \operatorname*{argmax}_{T\in\mathcal{T}} \mathcal{E}\left[(1-\lambda)|T\bigcap\tau| + \lambda(|T\bigcap\tau| - |T|)\right] \\
&= \operatorname*{argmax}_{T\in\mathcal{T}}(1-\lambda)\mathcal{E}\left[|T\bigcap\tau|\right] + \lambda\mathcal{E}\left[|T\bigcap\tau| - |T|\right] \\
&= \operatorname*{argmax}_{T\in\mathcal{T}} \left[(1-\lambda)\sum_{X\in T}\gamma(X)\right] + \left[\lambda\sum_{X\in T}(\gamma(X)-1)\right] \\
&= \operatorname*{argmax}_{T\in\mathcal{T}} \sum_{X\in T}(\gamma(X)-\lambda) \qquad (4.8)
\end{aligned}
$$

The result is a combined metric for balancing precision and recall. Note that, if λ=0, Equation 4.8 is the same as Equation 4.6 and thus maximizes labeled recall; and if λ=1, Equation 4.8 is the same as Equation 4.7 and thus maximizes labeled precision. Thus, λ functions as a mixing factor to balance recall and precision.

If we use this precision/recall tradeoff method to recombine n-best lists of base phrase trees, we can set λ to prefer precision over recall. Doing so will produce trees that only include a small number of high-certainty constituents and leave the remainder of the string unconstrained, regardless of whether such "high-certainty" trees were candidates in the original n-best base-phrase lists.

Figure 4.6 shows the results of performing this precision/recall tradeoff method on three separate n-best lists: the 50-best list of base-phrase trees extracted from the full-parse output of the Charniak/Johnson reranker [44]; the 50-best list output by the CSLUt shallow parser [109]; and the weighted combination of the two lists at various values of λ in Equation 4.4. For the combination, we set α=2 in Equation 4.1, with the reranker providing P_1, effectively giving the reranker twice

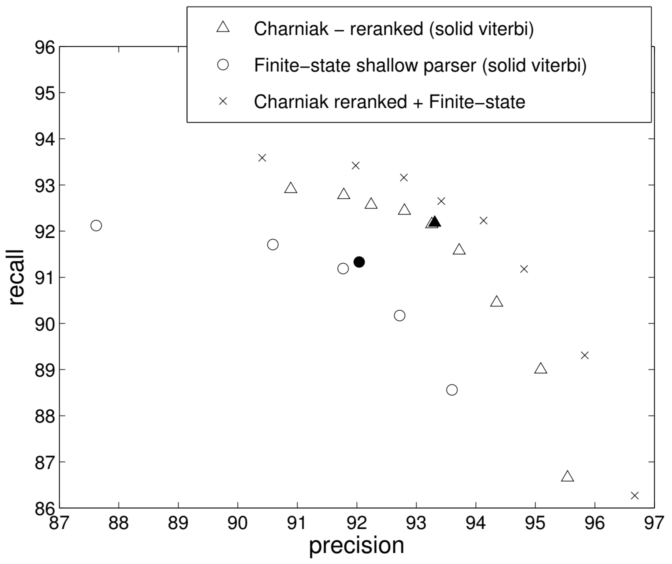


Figure 4.6: The tradeoff between recall and precision using a range of λ values (Equation 4.4) to select high-probability annotations from an n-best list. Results are shown on 50-best lists of base-phrase parses from two parsers, and on the combination of the two lists.

the weight of the shallow parser in determining the posteriors. The shallow parser has perceptron scores as weights, and the distribution of these scores after a softmax normalization was too peaked to be of utility, so we used the normalized reciprocal rank of each candidate as P_2 in Equation 4.1.

We point out several details in this graph. First, using this method does not result in an F-measure improvement over the Viterbi-best base-phrase parses (shown as solid symbols in the graph) for either the reranker or the CSLUt shallow parser. Also, using this model effects a greater improvement in precision than in recall, which is unsurprising with these non-hierarchical annotations; unlike full parsing (where long sequences of unary productions can improve recall arbitrarily), in base-phrase parsing, any given span can have only one non-terminal. Finally, the combination of the two n-best lists improves over either list in isolation; despite the fact that the CSLUt shallow parser consistently underperforms as compared to the full-parser reranker, the CSLUt output still contributes to improvements in the combination of the two systems.

4.5 Combination-Constrained Results

In this section we experiment with two different methods of combining constraints from different sources.

4.5.1 Recombined Constraints

In this section we explore the utility of constraining our parsing pipeline with a combination of the output from the shallow parser and the reranker, combined using the techniques outlined in Section 4.4.2. The recombined constraints condition is designed to compare the effects of generating constraints with different precision/recall characteristics, i.e., different λ parameters in Equation 4.4. For this experimental condition, we extract base-phrase trees from the n-best full-parse trees output by the reranker. We then *recombine* this list with the n-best list output by the CSLUt shallow parser, exactly as described in Section 4.4.2, again with the reranker providing P_1 and α=2 in Equation 4.1. The pipeline for this condition is shown in Figure 4.7. We examined

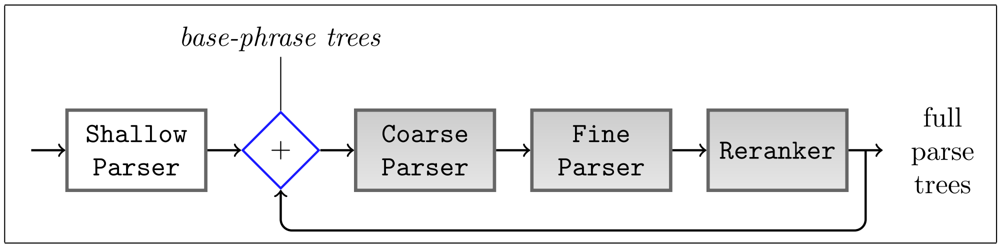


Figure 4.7: Recombining constraints from iterated and non-iterated sources.

a range of operating points from λ=0.4 to λ=0.9, and report two points here (λ=0.5 and λ=0.9), which represent the highest overall accuracy and the highest precision, respectively.

We see from Table 4.3 that combining the two base-phrase n-best lists to derive the constraints provides significant improvements when λ=0.5, to a total improvement of 0.9 and 0.5 percent over parser-best and reranker-best accuracy, respectively. We also see that performance degrades at λ=0.9 relative to λ=0.5. There are two interesting conclusions to be drawn from this result. Firstly, the λ=0.9 condition reaches a higher level of precision than the λ=0.5 condition, by design, as shown in the precision column (P) of Table 4.3. We had originally hypothesized that using just high-precision (high-confidence) constraints would result in improved downstream performance. Secondly, recall that optimizing for precision (by setting λ close to 1) will only allow a small number of constituents to be included in the constraint set. Therefore, the λ=0.5 condition actually places *more* constraints on a downstream search space than the λ=0.9 condition. This performance degradation at λ=0.9, then, tells us that, even at a lower precision, having more constraints is beneficial to downstream performance.

Constraints	Parser Best	Reranker Best	Oracle Best	Base-Phrase R	Base-Phrase P	Base-Phrase F
Baseline (Unconstrained)	88.9	90.2	**96.0**	–	–	–
Combo-constrained (λ=0.5)	**89.8**	**90.7**	95.4	92.2	94.1	**93.2**
Combo-constrained (λ=0.9)	89.3	90.4	95.9	81.0	**97.4**	88.4

Table 4.3: Full-parse F-scores on WSJ section 24. The unconstrained search (first row) provides a baseline comparison for the effects of constraining the search space. The bottom two rows demonstrate the effect of recombined constraints; the last three columns show the recall (R), precision (P), and F-score (F) of the constraints produced for each condition.

4.5.2 Unioned Constraints

When making a second pass through this pipeline, the original n-best list of full parses, output from the unconstrained parser, is available at no additional cost. Thus, our final set of experimental conditions investigate taking the union of constrained and unconstrained n-best lists. Imposing base-phrase constraints can result in candidate sets that are largely (or completely) disjoint from the unconstrained sets, and it may be that the unconstrained set is in many cases superior to the constrained set. Even our high-precision constraints did not reach 100% precision (as shown in Figure 4.6 and in Table 4.3), attesting to the fact that there was some *error* in all constrained conditions. By constructing the union of the two n-best lists, we can take advantage of the new constrained candidate set without running the risk that the constraints have resulted in a degraded n-best list. Figure 4.8 shows a union pipeline constructed for our next set of experiments; note where the full-parses output by the unconstrained parser in the first pass (top) is unioned with the constrained output in the second pass (bottom). We take a straightforward set union of the two n-best lists, and since the parser probabilities are produced from the same model in both passes,

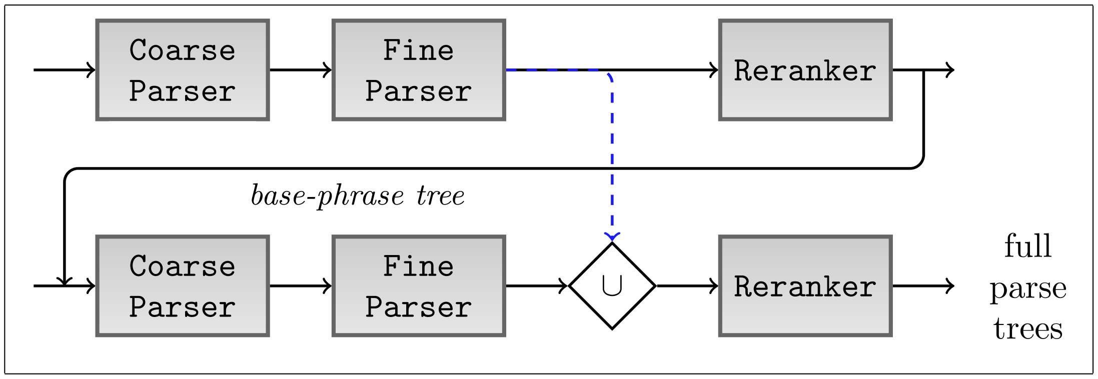


Figure 4.8: Unioning an n-best list of parses output by an unconstrained parser in the first pass through the pipeline (top) with an n-best list of parses output by a constrained parser in the second pass (bottom); the unioned list is then input to the reranker at the final stage.

the scores of each candidate in the two n-best lists are directly comparable.

Table 4.4 shows the results when taking the union of the constrained and unconstrained lists prior to reranking. Several interesting points can be noted in this table. First, note that taking the union results in an increase in parser-best accuracy over the baseline unconstrained list, under all constraints conditions. This difference tells us that the base-phrase constraints did cause some *search errors*, but also *resolved* other search errors. Recall that search errors occur when there is a higher-scoring candidate (according to the model) excluded from the search space. By looking at the unioned list of constrained and unconstrained parses, we can see that the parsing model did indeed rank some of the unconstrained parse candidates higher. If the parser-best F-score of the unioned lists was exactly the same as the unconstrained lists, then we would know that the parsing model had ranked all of the constrained parses lower than the unconstrained, which is clearly not the case. Conversely, we also know that the parsing model did not rank all of the constrained parses higher than the unconstrained, since the parser-best F-score of the unioned list differs from the constrained list. Since the unioned F-scores fall in-between the constrained and unconstrained set, we know that in some cases the base-phrase constraints prevented the parsing model from selecting its preferred candidates: thus, the constraints caused search errors. In other cases, the base-phrase constraints resolved search errors caused in the parsing model itself; the parser itself imposes a number of internal search constraints [34, 41, 44, 26], so the base-phrase constraints can actually correct for those internal-search errors by focusing the search in other areas. Taking the union of the constrained and unconstrained parses allows us to recover from the search errors; high-scoring full-parse candidates which were excluded by either the base-phrase constraints or the parser-internal search constraints are all included in the unioned n-best list for input to the reranker.

Second, by comparing the results from this table to those in Table 4.2, we can see that the parser-best F-score of the unioned lists is lower than the output from the constrained iteration alone. This tells us that the base-phrase constraints corrected *model errors*, where the highest-scoring candidate (according to the model) is not the best candidate (according to the reference). Model errors occur frequently, as is apparent from the difference between the oracle-best F-score and either the parser-best or reranker-best F-scores in Tables 4.2 and 4.4. However, the fact that the parser-best accuracy decreased after taking the union, but remains higher than the unconstrained accuracy, tells us that some of the constrained candidates with lower parser-probability are actually more accurate than competing unconstrained candidates with higher parser-probability. Thus, by using the constraints to prevent the parser from selecting its preferred candidate we were able to guide the search area such that the parser selected higher-accuracy candidates.

Constraints	**Parser Best**	**Reranker Best**	**Oracle Best**	**# Cand.**
Baseline (Unconstrained, 50-best)	88.9	90.2	96.0	47.9
Unconstrained ∪ Reranker-constrained	89.2	90.6	96.5	70.3
Unconstrained ∪ CSLUt-constrained	89.4	90.3	96.6	74.9
Unconstrained ∪ Combo (λ=0.5)	89.3	90.8	96.5	69.7
Unconstrained ∪ Combo (λ=0.9)	89.0	90.4	96.4	62.1
Unconstrained (100-best)	88.8	90.1	96.4	95.2
Unconstrained (50-best, beam×2)	89.0	90.5	96.1	48.1

Table 4.4: Full-parse F-scores on WSJ section 24 after taking the set union of unconstrained and constrained parser output under the 4 different constraint conditions. Also, F-score for 100-best parses, and 50-best parses with an increased beam threshold, output by the Charniak parser under the unconstrained condition.

Third, let us address the results of using the CSLUt (non-iterated) constraints in row 3 of Table 4.4. Despite the fact that the CSLUt-constrained condition hurts performance in Table 4.2, the union provides a 0.5 percent improvement over the baseline in the parser-best performance. Surprisingly, even though this set provided the highest parser-best and oracle-best F-score of all of the union sets, it did not lead to significant overall improvements after reranking. This is a puzzling result: by conventional wisdom, the highest-accuracy constraints (according to the mid-pipeline intrinsic evaluation) and/or the dataset with the highest oracle rate *should* result in the best pipeline-final solution. Thus we suspect that in order to predict effects on downstream performance, we will need to consider more than just the accuracy rates of a dataset. Chapter 6 will explore and expose some generally accepted ideas about accuracy and oracle rates in pipeline systems. Chapters 7 and 8 will examine several other characteristics of constraints in a pipeline system, and we will find that some of these characteristics explain this puzzling result; furthermore, we will also present techniques to correct for these detrimental characteristics.

Finally, the number of unique candidates in the unioned lists, as shown in the last column of Table 4.4, also provide some clues as to the effects of the constraints under the different conditions. The CSLUt-constrained unioned list contains the largest number of unique parse candidates, indicating that the CSLUt constraints resulted in a set of candidates that is the most disjoint from the original unconstrained parser. This result is perhaps unsurprising, since all of the iterated constraints use, to some extent, information already influenced by the unconstrained parser output. However, we might have expected that the more "diverse" set of candidate would have improved downstream performance, which was not the case here. Chapter 7 will examine this notion of diversity in a constraint set more systematically to determine its effects. The λ=0.5 and λ=0.9 unioned lists differ in size by about 7.5 candidates; the λ=0.9 condition results in fewer candidates under the union, though still more than in the original unconstrained list. There are fewer constraints in the high-precision condition, so the resulting n-best lists do not diverge as much from the original lists, leading to less of a difference between the unioned list and the constrained and unconstrained lists. However, even these few constraints do result in a change in the output of the parser.

In order to demonstrate that the gains in performance should not be attributed to increasing the number of candidates nor to allowing the parser more time to generate the parses, we included two more unconstrained baseline systems, shown in the last two rows of Table 4.4. The penultimate row in the table shows the results with 100-best lists output in the unconstrained condition, which does not improve upon the 50-best performance, despite an improved oracle F-score. We also compare against output obtained by doubling the beam threshold of the coarse-parser, since the second iteration through the parsing pipeline clearly increases the overall processing time by a factor of two. As shown in the last row of Table 4.4, doubling the beam threshold yields an insignificant improvement over the baseline despite a large processing burden.

We applied our best-performing model (Unconstrained $\cup$ Combo, λ=0.5) to the test set, WSJ section 23, for comparison against the baseline system. Our mixed-constraint, unioned-parses pipeline improved over the baseline on WSJ 23 by 0.4 absolute F-score; from 91.1 to 91.5, which is statistically significant at $p < 0.001$, using the stratified shuffling test [215].

4.6 Multiple Iterations and Convergence

The pipeline iteration technique introduced in this chapter could easily be repeated for *multiple iterations*. In fact, we experimented with using the output from the second pass of the pipeline to constrain a third pass, but found that this additional iteration did not yield further improvements. However, something can be said about the convergence properties of an iterated pipeline.

An iterated parsing pipeline, set up as described in this chapter with base phrases as constraints input to the context-free parser which generates n-best lists input to the final reranking stage, will converge in at most w iterations, where w is the length of the input string. To see that this is the case, consider the following:

1. Base phrase constraints as implemented in this chapter are *hard* constraints; once a base phrase is selected as a constraint, that phrase must necessarily occur in every parse output by the parser (and thus by the reranker as well).

2. If a base phrase occurs in every parse in an n-best list, then the conditional probability of that base phrase is 1. Regardless of the parameters selected for the precision/recall tuning algorithm, a constraint with probability 1 will be included in the constraint set.

3. Thus, base phrase constraints, once selected, will be retained in each successive iteration through the pipeline.

Furthermore,

1. Each base phrase must cover at least one word in an input sentence, and each word can be covered by at most one base phrase, since base phrases are non-hierarchical and non-overlapping.

2. Thus, the maximum number of base phrase constraints produced for any given sentence is w, where w is the length of the sentence.

3. If just one base phrase constraint were added after each iteration, then the pipeline would converge in w iterations.

This proof places an upper bound on the maximum number of iterations required to converge; in practice, base phrases typically cover more than one word, and do not cover every word in a sentence. Note that we make no guarantees about the empirical performance that might be observed at convergence, only that the constraint set extracted from the pipeline-final stage will remain constant after at most w iterations.

4.7 Conclusion

In this chapter we implemented an iterated parsing pipeline, and demonstrated that the generic technique of pipeline iteration can be useful in improving system performance, by constraining early stages of the pipeline with output derived from later stages. In Section 4.3 we compared the results of non-iterated constraints with iterated constraints, and found that the iterated constraints typically performed better. In Section 4.4.2 we used the precision/recall tradeoff method from Goodman's [100] thesis allowed us to vary the amount of constraints placed on the early stages of the pipeline. The results in Section 4.5.1 showed that heavily-constrained parses outperformed the less-constrained parses from using only high-confidence (high-precision) constraints.

Our best results were achieved by taking the union of unconstrained parses and relatively heavily-constrained parses as input to the final reranking stage of the pipeline. However, taking the union of n-best lists also allowed us to analyze whether the base-phrase constraints were affecting search errors and model errors in the pipeline. In Section 4.5.2 we found that, indeed, the constraints did cause some search errors but *also* resolved other search errors caused by internal thresholds in the parsing model itself. We also determined that the constraints were also able to resolve model errors in the parser. Thus this work has shown that constraints are not necessarily harmful to the accuracy of a pipeline.

While the current work made use of a particular kind of constraint—base phrases—many others could be extracted as well. Preliminary results extending the work presented in this chapter show parser accuracy improvements from pipeline iteration when using constraints based on an unlabeled partial bracketing of the string. Higher-level phrase segmentations or fully specified trees over portions of the string might also prove to be effective constraints. Furthermore, the techniques shown here are by no means limited to parsing pipelines, and could easily be applied to pipeline systems in other fields.

Chapter 5

Model Interaction

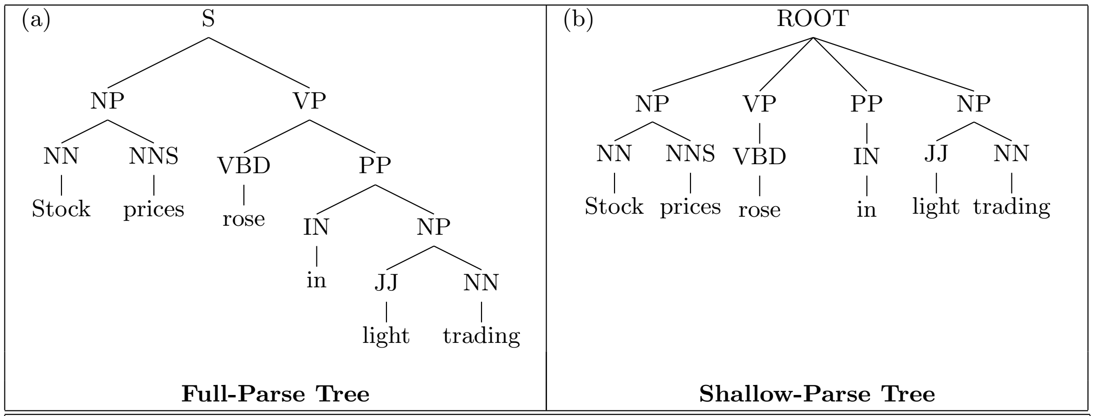
In the previous chapter we explored the utility of prefixing a finite-state parser onto the beginning of a context-free parsing pipelines. This technique, of placing a model of lower complexity upstream of a higher-complexity model in the pipeline, is the traditional method for constructing a pipeline. And in fact, this technique is necessary to maintain the efficiency benefit of the pipeline architecture. However, in this chapter[1] we will upset this accepted practice, by exploring the effects of placing a higher-complexity (context-free) model upstream of a lower-complexity (finite-state) model. Our results will demonstrate how much accuracy is sacrificed for efficiency in more traditional architectures, establishing that combining the output of context-free and finite-state parsers gives much higher results than the previous-best published results, on several common shallow parsing tasks.

In addition to demonstrating the accuracy benefits that can be achieved by combining these two parsers, we will also experiment with different methods for system-combination and their effects on pipeline performance. Specifically, we will examine: (1) generating a constraint space as n-best sequences or lattices; (2) weighted and unweighted system combination; and (3) test-time versus train-time constraints. The traditional argument is that when training a statistical model, one must match the training conditions to the testing conditions (*c.f.* [184]). Thus, for example, in training a perceptron model (discussed in Section 3.1.5, p. 29), we use unseen data as heldout during training, to replicate the unseen test data and to prevent over-training on the training data [65]. Cross-validation is performed to ensure that the models are not tested on training data. However, in this chapter we demonstrate surprising results wherein altering the test-time conditions from train-time conditions had a positive impact on performance. Much of the work in subsequent chapters of this dissertation is motivated by these surprising results, to understand the results as a step towards understanding pipeline systems.

5.1 Finite-State vs Context-Free Parse Trees

Finite-state parsing (also called chunking or shallow parsing) has typically been motivated as a fast first-pass for—or approximation to—more expensive context-free parsing [1, 172, 2]. Finite-state parsing provides useful syntactic annotations of text (although the annotations may not be as rich as those obtained from a context-free parser). Figure 5.1 demonstrates the different between (a) a full parse tree and (b) a shallow parse tree. Note that the shallow parse tree is non-hierarchical, and thus can be represented as a sequence of tags, as shown in Figure 5.1(c). Here we follow established conventions [172, 207, 194] of marking each word as either beginning a chunk ('B'), being inside of

[1] Part of the work in this chapter was published in EMNLP'05 [109].



(c)

chunk tags:	B-NP	I-NP	B-VP	B-PP	B-NP	I-NP
POS-tags:	NN	NNS	VBD	IN	JJ	NN
words:	Stock	prices	rose	in	light	trading

Shallow-Parse Tagged Sequence

Figure 5.1: The full-parse tree (a) and shallow-parse tree (b) for an example sentence. The shallow-parse tree can also be represented as the tagged sequence shown in (c), allowing for finite-state processing.

a chunk ('I'), or being outside of any chunk ('O'). For tasks which use multiple chunk-types, i.e., noun phrases (NP), verb phrases (VP), prepositional phrases (PP) etc., the chunk-type is affixed to the chunk tag label, as shown in the figure.

Since finite-state parsing is many orders of magnitude faster than context-free parsing, finite-state parsing can provide useful syntactic annotations for large amounts of text in cases where context-free parsing is too expensive (e.g. for many very-large-scale natural language processing tasks such as open-domain question answering from the web). For this reason, finite-state parsing has received increasing attention in recent years. In contrast to much of the other research on finite-state parsing, here we argue that rather than using a finite-state parser to approximately replace a context-free parser, it can be more beneficial to combine the two parsers. We begin in the next section by discussing the comparison of finite-state and context-free parsers.

5.2 Model Comparison

In addition to the clear efficiency benefit of finite-state parsing, Li and Roth [144] further claimed both an *accuracy* and a *robustness* benefit of finite-state parsing over context-free parsing. Li and Roth demonstrated that their finite-state parser, trained to label shallow chunks along the lines of the well-known CoNLL-2000 Chunking task [207], outperformed the Collins context-free parser in correctly identifying these chunks in the Penn Wall Street Journal (WSJ) Treebank [150]. They argued that their superior performance was due to optimizing directly for the local sequence labeling objective, rather than for obtaining a hierarchical analysis over the entire string.

In point of fact, as we demonstrated in [109], the difference in parser performance was due to an erroneous assumption in `chunklink`, the conversion script used to convert the full parses output by context-free parsers into shallow parses for comparison to finite-state parsers. In that paper we demonstrated that changes to the conversion routine, which take into account differences between the original treebank trees and the trees output by context-free parsers (in particular, a reliance on empty nodes), eliminated the previously-reported accuracy differences. We also showed that a convention that is widely accepted for evaluation of context-free parses—ignoring punctuation when setting the span of a constituent (see Section 3.3.2 p. 41)—results in improved shallow parsing performance by certain context-free parsers across a variety of shallow parsing tasks. Thus we refuted Li and Roth's [144] argument, and found instead that under a fair evaluation scenario, the performance of finite-state parsers and context-free parsers is nearly identical on shallow parsing tasks, as will be shown in the following section.

5.2.1 Shallow Parsing Tasks

In this section we will compare the performance of our CSLUt finite-state parser and the Charniak and Johnson [44] context-free parser on three finite-state parsing tasks, described below. As will be shown, these two parsers perform nearly identically on all three tasks, despite the previously reported comparisons of context-free parsers and finite-state shallow parsers [144] which greatly under-estimated the performance of context-free parsers.

Two commonly reported shallow parsing tasks are Noun-Phrase (NP) Chunking [172] and the CoNLL-2000 Chunking task [207]. The NP-Chunking task, originally introduced by Ramshaw and Marcus [172] and also described by Collins [65], Sha and Pereira [194], brackets just base NP constituents.[2] The CoNLL-2000 task, introduced as a shared task at the CoNLL workshop in 2000 [207], extends the NP-Chunking task to label eleven different base phrase constituents annotated in the Penn Treebank, including: ADJP, ADVP, CONJP, INTJ, LST, NP, PP, PRT, SBAR, UCP and VP. Anything not in one of these base phrases is designated as "outside" (O). Reference shallow parses for this latter task were derived from treebank trees via a conversion script known as `chunklink`.[3]

We follow Li and Roth [144] in using `chunklink` to also convert the full-parse trees output by a context-free parser into the flat tag sequences representing shallow-parse trees. However, we additionally pre-processed the output of the Charniak and Johnson [44] context-free parser to correct for the errors in `chunklink`; see [109] for full details.

For both the NP-Chunking and the CoNLL-2000 Chunking tasks, the training set is sections 15-18 of the Penn WSJ Treebank and the test set was section 20. We follow Collins [65] and Sha and Pereira [194] in using section 21 as a heldout set. The third task, introduced by Li and Roth [144] and which we term the *Chunking-XL task*, performs the same labeling as in the CoNLL-2000 Chunking task, but with more training data and different testing sets: training was WSJ sections 2-21 and test was section 00. We use section 24 as a heldout set, which is often used as heldout for training context-free parsers. Training and testing data for the CoNLL-2000 task is available online.[4] For the heldout sets for each of these tasks, as well as for all data sets needed for the Chunking-XL task, reference shallow parses were derived from treebank trees via the `chunklink` conversion script. All data was tagged with the Brill [28] POS tagger after the `chunklink` conversion. We verified that using this method on the original treebank trees in sections 15-18 and 20 generated data that is identical to the CoNLL-2000 data sets online; replacing the POS tags in the input text with Brill POS tags before the `chunklink` conversion results in slightly different shallow parses.

Table 5.1 shows several state-of-the-art results[5] on three shallow parsing tasks. We can see from the table that CSLUt is fairly competitive on all three shallow parsing tasks. The table shows that

[2] We follow Sha and Pereira [194] in deriving the NP constituents from the CoNLL-2000 data sets, by replacing all non-NP shallow tags with the "outside" ("O") tag. They mention that the resulting shallow parse tags are somewhat different than those used by Ramshaw and Marcus [172], but that they found no significant accuracy differences in training on either set.

[3] http://ilk.kub.nl/∼sabine/chunklink/.

[4] http://www.cnts.ua.ac.be/conll2000/chunking/

[5] Sha and Pereira [194] reported the Kudo and Matsumoto [132] performance on the NP-Chunking task to be 94.4 and to be the best reported result on this task. In the cited paper, however, the result is as reported in our table.

System		NP-Chunking	Chunking	Chunking-XL
Ando and Zhang [4]		94.7	94.4	–
Kudo and Matsumoto [132]		94.2	93.9	–
Li and Roth [144]		–	93.0	94.6
Sha and Pereira [194]	CRF	94.4	–	–
	voted perceptron	94.1	–	–
Sutton et al. [203]		94.8	–	–
CSLUt [109]		94.2	93.5	95.1
C&J [44]	parser-best	94.2	93.8	95.2
	reranker-best	94.8	94.3	95.8

Table 5.1: F-measure shallow bracketing accuracy on three shallow parsing tasks, for several competitive finite-state parsers and the Charniak and Johnson [44] context-free parser (C&J). Results reported in this table include the best published results on each of the three shallow-parsing tasks.

the Charniak and Johnson [44] parser and reranker are clearly also competitive on these tasks: the parser-best output is only 0.6 percentage points below the best-published result on the Chunking task, and the reranker output improves quite a bit over the parser

We can also see from Table 5.1 that performance levels of the CSLUt finite-state parser and the Charniak and Johnson [44] context-free parser are very similar for all three tasks, while the reranking model outperforms both. Despite having very similar performance, however, it is still possible that because finite-state and context-free parsers are optimized for different objectives, their output predictions might differ in a complementary way. It is likely true that a context-free parser which has been optimized for global parse accuracy will, on occasion, lose some shallow parse accuracy to satisfy global structure constraints that do not constrain a shallow parser. However, it is also likely true that these longer distance constraints will on occasion enable the context-free parser to better identify the shallow constituent structure. Thus the next section will explore methods to exploit the complementary nature of parsers optimized for different objectives by exploring different methods for combining output from finite-state and context-free parsers.

5.3 Model Intersection

In Chapter 4 we looked at a simple union of data produced by different models. In this section we will look at the effects of *intersecting* two data sets, where one data set is derived from a context-free model and the other from a finite-state model.

For the experiments described in this section, consider the CSLUt finite-state parser as a reranker of the context-free parser output, with the objective of improving shallow parse accuracy. In all results reported below, we will be incorporating the k-best context-free parser output into the finite-state parsing model. Note that when the 50-best lists of full-parses output by the Charniak parser were converted into k-best lists of shallow parses using `chunklink`, many of the context-free parses map to the same finite-state parse, so the size of the set of shallow parses output by the Charniak parser is typically much less than 50, with an average of around 7. In addition to providing a list of candidates, the context-free parser also outputs the log probability of each candidate according to its model. This probability score could either be discarded or used to scale the weights in the CSLUt finite-state model.

We begin by constructing the pipeline shown in Figure 5.2. Note that in this pipeline, the output of the Charniak context-free parser is input to the CSLUt finite-state parser. This is a reversal of the typical increase of model complexity as constraints are passed down the pipeline; thus pipeline is less efficient than other shallow parsers, but we will show that the loss in efficiency

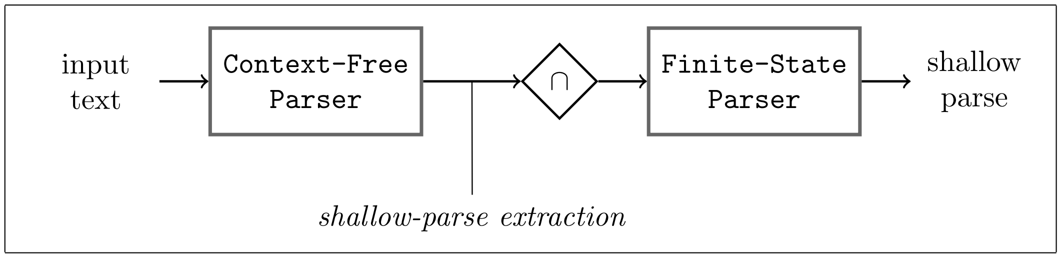


Figure 5.2: The Charniak/Johnson context-free parser constraining our CSLUt finite-state shallow parser. Shallow parses are extracted from the context-free parses using the corrected variant of `chunklink`. At the diamond decision point in the figure, intersection constraints (∩) will be applied, with or without weights, at train-time, test-time, or both, as appropriate.

is traded for a large gain in accuracy, higher than that of any previously-reported shallow parsing system. In fact, Sutton et al. [203] erroneously claim to have achieved highest-reported accuracy at levels lower than those we report here and in [109].

For our experiments in this section, the CSLUt shallow parser is unconstrained during training, then restricted to just the output from the Charniak parser during testing. Our first set of experimental conditions restrict the shallow parser to selecting one sequence in its entirety from among the k-best Charniak candidates.[6] We refer to this experimental condition as *sequence* constraints. In the second set of experiments, we relax the previous conditions slightly by connecting the k-best parses (output by the context-free parser) to generate a confusion network lattice, and allowing CSLUt to select any state sequence from the lattice; we will refer to this as *lattice* constraints.

Each of these experiments also explore the effects of weighted and unweighted intersection. We begin with *unweighted* constraints, which create the simplest kind of ‘rovered’ system [84], restricting the set of shallow parse candidates to the intersection of the sets output by each system, discarding the score of the upstream context-free parser. Since the Viterbi search of the CSLUt model provides a score for all possible shallow parses, the intersection of the two sets is simply the set of shallow-parse sequences in the 50-best candidates output by the Charniak parser. We then use the CSLUt perceptron-model scores to choose from among just these candidates. Effectively, we use the Charniak parser’s k-best shallow parses to limit the search space for our shallow parser at test-time.

Next we use *weighted* constraints, which extend the unweighted intersection by including the log probability scores from the context-free parser (*c.f.* [66]). The score for a shallow parse output by the Charniak parser is the log of the sum of the probabilities of all context-free parses mapping to that shallow parse. We normalize across all candidates for a given string, thus these are conditional log probabilities. We multiply these conditional log probabilities by a scaling factor α before adding them to the CSLUt perceptron score for a particular candidate. The best-scoring candidate using this composite score is selected from among the shallow parse candidates output by the Charniak parser. We used the heldout data to empirically estimate an optimal scaling factor for the Charniak scores, which is 15 for all trials reported here. This factor compensates for differences in the dynamic range of the scores of the two parsers.

Our third experimental condition is also a weighted intersection, but uses the exponential of the sum of weights output by the Charniak/Johnson reranker for a given parse. We refer to this experimental condition as *RR-weighted*. As in the second condition, the scores are normalized across all candidates. The optimal scaling factor for the reranker scores, again empirically estimated using the heldout data, was set to 30, indicating a strong reliance on the reranker score.

As mentioned previously, all of these intersections are done at test-time; each of the models

[6] Thus the CSLUt parser serves as a *selection* stage (discussed in Chapter 2, p. 19), similar to a reranker.

System		NP-Chunking	Chunking	Chunking-XL
CSLUt [109]		94.2	93.5	95.1
C&J [44]	parser-best	94.2	93.8	95.2
	reranker-best	94.8	94.3	95.8
Unweighted-sequences		94.6	94.3	95.6
Weighted-sequences		**95.2**	**94.8**	96.0
RR-weighted–sequences		95.1	94.6	96.0
Unweighted-lattice		94.6	94.1	95.6
Weighted-lattice		95.1	94.5	95.9
RR-weighted–lattice		**95.2**	94.6	**96.1**

Table 5.2: F-measure shallow bracketing accuracy on three shallow parsing tasks, for the CSLUt shallow parser, the Charniak and Johnson [44] context-free parser and reranker, and with unweighted and weighted combinations of the CSLUt and Charniak & Johnson (C&J) systems either at test-time, train-time, or both.

were trained independently, and only at test-time is the finite-state parser restricted to the space defined by the context-free parser. To remain consistent with task-specific training and testing section conventions, the individual models were always trained on the appropriate sections for the given task, i.e., WSJ sections 15-18 for NP-Chunking and the CoNLL-2000 tasks, and sections 2-21 for the Chunking-XL task.

Results from these methods of combination are shown in Table 5.2. Even the simple unweighted intersection gives quite large improvements over each of the independent systems for all three tasks, resulting in the highest-reported results of the Chunking and the Chunking-XL tasks. All of these improvements are significant at $p<0.001$ using the Matched Pair Sentence Segment test [94]. The parser-weighted intersection gives further improvements over the unweighted intersection for all tasks, and this improvement is also significant at $p<0.001$, using the same test. The reranker weights resulted in an improvement under the lattice-intersection condition but not the sequence-intersection. The best results for all three of the shallow parsing tasks (bolded in the table) utilize the upstream weight; we will discuss in Chapter 8 that a weighted set of constraints is often more beneficial than an unweighted set.

5.4 Model Training

In the previous section we observed that constraining a finite-state parser at test-time, with output from a context-free parser, resulted in output higher than either of the parsers achieved alone. One might wonder whether constraining the finite-state parser during training would also prove beneficial, or whether this would remove the complementary effects of training the two models for different objectives. Therefore in this section we will conduct a thorough evaluation of constraining a model during training, testing, or both.

First, however, allow us to take a short digression to motivate this and, indeed, several other experiments conducted for this dissertation. Recall that in the previous chapter we observed that the union of the CSLUt-constrained Charniak parser output and the unconstrained parser output resulted in the highest parser-best and oracle-best context-free parse trees (see Section 4.5.2). The relevant results from Chapter 4 are reproduced here, in the first three columns of Table 5.3. From these numbers we expected the 50-best $\cup$ CSLUt-constrained condition to be the best performer. However, the reranker-best output was disappointingly low.

McClosky et al. [152] reported a similar result, in which expectations of improved performance were not met by the reranker output. In that paper, which we discussed in Chapter 3 (p. 54), they only allude to this disappointing result in a footnote, noting that they "attempted to retrain the

Condition	**Parser-best**	**Reranker-best**	**Mis-trained RR-best**	
50-best	88.9	90.2	–	
100-best	88.8	90.1	**90.2**	
Self-trained	90.2	90.9	**91.1**	
CSLUt-constrained	88.4	89.5	89.3	
Reranker-constrained	89.6	90.5	90.5	
50-best ∪ CSLUt	89.4	90.3	**90.6**	*
50-best ∪ Reranker	89.2	90.6	90.5	

Table 5.3: Comparison of rerankers trained under matching train/test conditions (Reranker-best) and under mismatched train/test conditions (Mis-trained RR-best).

reranker" but no improvement was achieved. They then presented results using a reranker trained during an earlier iteration of their system. Following the example set by McClosky et al. [152], we use the reranking model trained on the (unconstrained) 50-best output, and apply it to each of our reported datasets. We refer to this technique as using an "mis-trained" reranker; results are shown in the final column of Table 5.3. Furthermore, we replicated the best-performing setup from [152], namely re-training the parser on the reranker-best parses on the North American News Text Corpus (NANC), mixed with five copies of sections 02-21 of the WSJ Treebank (reported as "WSJx5+1,750k" in that paper); we report results on this setup as the Self-trained condition. We also applied our mis-trained reranker model to the Self-trained output, also reported in Table 5.3. Note from the bolded results in the table that there are three conditions under which the mis-trained reranker outperforms the trained reranker model, although only the 50-best ∪ CSLUt-constrained condition resulted in a statistically significant improvement using using the Matched Pair Sentence Segment test [94].

These as-yet-unexplained results led us to ask whether the accepted paradigm of matching training and testing conditions might not be the most beneficial approach. We will explore this question in the remainder of this chapter; other chapters will return to these results again in an attempt to understand and explain this phenomenon.

5.4.1 Matched Train/Test Conditions

All of the experimental conditions explored in Section 5.3 could, as we alluded to earlier, be applied either during training or at test-time. For the train-time constraints, the CSLUt perceptron model will essentially use the context-free parser as the Gen function (see Section 3.1.5, p. 29 for an explanation of the perceptron model). Note that this is similar to the training scenario for the Johnson reranker, which is trained on cross-validation output from the Charniak parser.

The experiments conducted in this section are identical to those in Section 5.3 with the exception that in this section, we will *match* the training and testing conditions. Thus, if the CSLUt model is to be constrained to the sequences output by the Charniak parser at test-time, then CSLUt is also constrained to such sequences (produced in the typical cross-fold validation scenario) during training. Similarly for lattice constraints, and weighted sequence or lattice constraints. The results of these matched train/test experiments are shown in Table 5.4, along with the results from the each of the parsers alone.

Under these matched train/test conditions, the best results for each task (see the bolded numbers in the table) are obtained by training and testing the CSLUt finite-state parser on the weighted shallow-parse sequences output by the Charniak context-free parser. However, all of these results are lower than the best result seen in Table 5.2, and with the exception of the Chunking-XL task, none of these system intersections improved over the best single-system result.

System		**NP-Chunking**	**Chunking**	**Chunking-XL**
CSLUt [109]		94.2	93.5	95.1
C&J [44]	parser-best	94.2	93.8	95.2
	reranker-best	94.8	94.3	95.8
Train-time Unweighted-Sequences				
test-time unweighted-sequences		94.1	93.6	95.4
Train-time Weighted-Sequences				
test-time weighted-sequences		**94.8**	**94.3**	**96.0**
test-time rr-weighted-sequences		94.4	93.8	95.7
Train-time Unweighted-Lattice				
test-time unweighted-lattice		93.9	93.4	95.3
Train-time Weighted-Lattice				
test-time weighted-lattice		94.5	94.1	95.8
test-time rr-weighted–lattice		94.7	94.2	95.8

Table 5.4: F-measure shallow bracketing accuracy on three shallow parsing tasks, using matched train-time and test-time conditions to train the CSLUt shallow parser on the Charniak & Johnson (C&J) system output.

5.4.2 Mismatched Train/Test Conditions

In our next set of experiments, we will force a *mismatch* between the training and testing conditions. For example, if the CSLUt model is to be constrained to the sequences output by the Charniak parser at test-time, then CSLUt will *not* be constrained in the same manner at during training. The CSLUt model may be unconstrained during training, or trained on weighted sequence output. The results of these mismatched train/test experiments are shown in Table 5.5. In the top rows of the table we once again replicate the results from the each of the parsers alone, as well as the results from Table 5.2 which, after all, are also mismatched train/test conditions.

Under these mismatched train/test conditions, the best results for each task (see the bolded numbers in the table) are still those from the unconstrained training/constrained testing conditions from Table 5.2, though training on unweighted sequences and testing on weighted sequences is tied for the best performance on the Chunking-XL task. Note the sharp degradation in performance when allowing an unconstrained search at test-time in a training-constrained model.

The success of the train-time unconstrained/test-time constrained experimental results might suggest that perhaps it is better to have a relatively unconstrained training space (or alternately to let the training model "choose," or dictate, its own training space) and then only constrain the test space. We apply this concept to the Charniak and Johnson [44] context-free parsing pipeline to test its generality.

We have a couple of different options for applying this technique of mismatching train/test conditions to train the Johnson reranking model. While we cannot feasibly train the Johnson reranking model on an unconstrained space due to computational limitations,[7] we could train it on a much larger number of candidates than we test it on. We could also do the reverse, by training the model on a much smaller number of candidates than used at test-time.

Table 5.6 demonstrates the effects of restricting the reranker's search space to varying degrees during training and testing. For each of these experiments, the Johnson reranker was trained on crossfold-validated n-best parses output on WSJ 02-21 with section 00 as heldout. The italicized numbers along the diagonal represent matched train/test conditions, i.e., the traditional experimental conditions. In this table we do not observe the sharp degradation that we saw in the

[7] Ironically, reducing the search space for highly-complex, computationally intense models has long been the impetus for implementing a pipeline system.

System		**NP-Chunking**	**Chunking**	**Chunking-XL**
CSLUt [109]		94.2	93.5	95.1
C&J [44]	parser-best	94.2	93.8	95.2
	reranker-best	94.8	94.3	95.8
Train-time Unconstrained				
test-time unweighted-sequences		94.6	94.3	95.6
test-time weighted-sequences		**95.2**	**94.8**	96.0
test-time rr-weighted-sequences		95.1	94.6	96.0
test-time unweighted-lattice		94.6	94.1	95.6
test-time weighted-lattice		95.1	94.5	95.9
test-time rr-weighted–lattice		**95.2**	94.6	**96.1**
Train-time Unweighted-Sequences				
test-time (unconstrained)		82.4	79.7	87.4
test-time weighted-sequences		95.1	94.6	**96.1**
test-time rr-weighted-sequences		94.8	94.4	95.9
Train-time Weighted-Sequences				
test-time (unconstrained)		46.1	42.4	67.8
test-time unweighted-sequences		91.3	90.9	93.9
Train-time Unweighted-Lattice				
test-time (unconstrained)		82.4	84.5	89.5
test-time weighted-lattice		94.9	94.4	95.9
test-time rr-weighted–lattice		95.1	94.6	96.1
Train-time Weighted-Lattice				
test-time (unconstrained)		46.1	19.1	38.8
test-time unweighted-lattice		89.9	85.4	89.2

Table 5.5: F-measure shallow bracketing accuracy on three shallow parsing tasks, using mismatched train-time and test-time conditions to train the CSLUt shallow parser on the Charniak & Johnson (C&J) system output.

shallow-parsing task, where training on a small, highly-constrained dataset resulted in very low performance when testing on a larger (less-constrained) dataset. However, we can see that the unconstrained-train/constrained-test technique was also successful for context-free parse reranking: using the largest, least-constrained training set of 1000-best parses[8] resulted in the highest F-scores for every size of test set.

	Test-time			
Train-time	**10-best**	**50-best**	**100-best**	**1000-best**
10-best	*89.9*	89.9	89.9	89.9
50-best	**90.2**	*90.2*	90.2	90.4
100-best	**90.2**	90.3	*90.1*	90.4
1000-best	**90.2**	**90.4**	**90.3**	***90.6***

Table 5.6: Reranker-best F-scores on WSJ section 24 full-parses output by the Charniak and Johnson [44] parser/reranker under matched (italicized) and mismatched train/test conditions. Bolded numbers indicate the best-performing training condition for each given testing condition.

[8] In order to train on such large n-best lists and their correspondingly large feature sets, we modified the feature representation in the reranker to consume far less memory than required by the out-of-the-box vector representation; our modifications will be made available online.

5.5 Conclusion

In this chapter we examined interactions between models in a pipeline system. We demonstrated that there is no accuracy or robustness benefit to shallow parsing with finite-state models over using high-accuracy context-free models, in contrast to what had been previously reported. Furthermore, there is a large benefit to be had in combining the output of high-accuracy context-free parsers with the output of shallow parsers, resulting in a large improvement over the previous-best reported results on several tasks, including the well-known NP-Chunking and CoNLL-2000 shallow parsing tasks. Thus, strictly enforcing the ordering of model complexity in a pipeline, as is typically done, can prevent one from producing a state-of-the-art result. We also saw that enforcing a match between train- and test-time conditions can be detrimental, and that sometimes, a train/test mismatch can result in the highest levels of performance. In the final section of this chapter, we observed the curious results of applying a "mis-trained" reranking model—one that had been trained under different conditions than it was tested on—where the mis-trained model outperformed the trained model in certain cases. The take-away pipeline technique from this chapter is to challenge the traditions in pipeline construction, particularly in terms of model complexity and matching train/test conditions.

Part III

Constraint Characteristics

The search space in an NLP pipeline system could be represented as a multi-dimensional space, with each data point in the space representing a possible solution. Such a space, constrained by the output of an upstream stage, might be *characterized* in many different ways. The space might be large or small, densely or sparely populated, with rigid or fuzzy boundaries, and with peaks and valleys of varying height. Current practices generally neglect to analyze the characteristics of the search space as a whole, focusing instead on only one or two (if any) aspects of the space. This disregard for the search space as a whole can be seen in the literature: the reported characteristics of a fully-specified constrained search space [44, 103, 134, 152, 175] include the accuracy of the one-best and oracle-best candidates, but ignore other characteristics such as the average accuracy of the candidates in the set or how completely the candidates cover the space. In pipeline systems implementing partial-solution constraints [22, 56, 79, 88, 146, 190], the accuracy of the constraints are reported, but not, for example, the restrictiveness of the constraint set.

Thus, although a few characteristics of the search space have been reported in the literature of pipeline systems, others that would be necessary to describe the search space as a whole, are simply not mentioned. The next three chapters of this dissertation will discuss characteristics of a constrained space, including of course the accuracy of the constraints in the space (Chapter 6), but going beyond one-best and oracle-best metrics; the diversity, regularity, density, and coverage of such a space (Chapter 7); and the peaks and valleys of the space as defined by a probability distribution over the space (Chapter 8). Each of these chapters will present quantitative metrics for measuring these spatial characteristics, as well as the empirical effects on pipeline performance when these characteristics are altered.

Chapter 6

Constraint Accuracy

In this chapter we report on the most obvious and most-cited characteristic of a constraint set: the accuracy of the constraints. We will argue that the two oft-reported measures of one-best and oracle-best accuracy are not sufficient to characterize the quality of constraints. An improvement in the oracle-best and one-best rates of constraining input does not always correspond to improvements in the output. As we saw in Chapters 4 and 5, a decreased oracle in the constraints does not necessarily correspond to decreased accuracy of the output—and vice versa. Similarly, an increase in mid-pipeline, intrinsic accuracy does not necessarily correspond to an increase in the pipeline-final extrinsic accuracy. Thus the point of this chapter is to introduce several accuracy metrics and demonstrate that these metrics are insufficient for predicting downstream pipeline performance.

6.1 Accuracy Metrics

In NLP, two intrinsic evaluation metrics are often reported: *one-best* accuracy, i.e., the accuracy of the best solution according to the prior distribution defined over the search space, and *oracle-best* accuracy, the accuracy of the best-possible solution in the defined search space. In this chapter we discuss these metrics and introduce two new ones: *oracle-worst* accuracy, and *self-reference* accuracy.

6.1.1 One-Best

Measuring the accuracy of the one-best constraint, where the constraints are ordered by model-score, is a common method used to estimate the quality of pipeline constraints. Improvements in one-best accuracy of an intrinsic evaluation does not necessarily correspond to improvements in the overall (extrinsic) accuracy of the pipeline.

Table 6.1 demonstrates how the one-best accuracy output by the parser compares to the pipeline-final accuracy output by the reranker. While the Self-trained condition has both the

Condition	**Parser Best**	**Reranker Best**	**Diff**
50-best	88.9	90.2	1.3
100-best	88.8	90.1	1.3
Self-trained	90.2	90.9	0.7
CSLUt-constrained	88.4	89.5	1.1
Reranker-constrained	89.6	90.5	0.9
50-best ∪ CSLUt	89.4	90.3	0.9
50-best ∪ Reranker	89.2	90.6	1.4

Table 6.1: Comparison of one-best parser accuracy rates and pipeline-final reranker accuracy rates.

Condition	n	Parser Best	Reranker Best	Oracle Best	Oracle Worst	Oracle Diff
50-best	47.9	88.9	90.2	96.0	68.1	27.9
100-best	95.2	88.8	90.1	96.4	64.6	31.8
Self-trained	49.8	90.2	90.9	96.4	67.0	29.4
CSLUt-constrained	46.2	88.4	89.5	94.1	69.1	25.0
Reranker-constrained	46.9	89.6	90.5	95.1	69.9	25.2
50-best $\cup$ CSLUt	75.0	89.4	90.3	96.6	65.6	31.0
50-best $\cup$ Reranker	72.1	89.2	90.6	96.5	66.4	30.1

Table 6.2: Comparison of two accuracy measurements of oracle-best and oracle-worst rates.

highest one-best accuracy and the highest reranker-best accuracy, note in the final column of the table that the difference between the parser-best and the reranker-best is lower than might be expected.

6.1.2 Oracles

Oracle rates are calculated by comparing each candidate in a set to the true, reference candidate and selecting the candidate with the highest accuracy[1] as the "oracle" candidate. In this section we will distinguish between *oracle-best* rates (typically referred to in the literature as *oracles*) and *oracle-worst* rates; an oracle-worst candidate is selected from a set as the candidate with the lowest accuracy.

Oracle-best rates are typically much higher than the one-best accuracy rates [174], as shown in Table 6.2. Arguments have been made that a higher oracle rate is always better. We argue here that if the difference between one-best and oracle-best accuracy (or between oracle-worst and oracle-best) is reduced but still remains fairly large, then a reduced oracle rate is not necessarily an indication that the overall accuracy of the pipeline will also be reduced. Table 6.2 demonstrates empirically that improvement in either *oracle-best* or *oracle-worst* does not directly correspond to improvement in reranker-performance.

As a Function of Size

The oracle rate of a system can often be increased by simply enlarging the search space. Certainly, a less-constrained, larger space may cause fewer search errors downstream. However, as we saw in Chapter 4, constraints can also correct for model errors; a less-constrained space will likely correct fewer model errors. We can see from Table 6.2 that the size of the candidate set does not necessarily correspond to the oracle rates; for example, the Self-trained condition has half the number of candidates as the 100-best condition but matches the oracle-best rate and has a higher oracle-worst rate.

An overly-constrained, smaller search space can be problematic as well. Clark and Curran [57, 59] found that, in training discriminative models, the number of (incorrect) solutions contained in the search space impacted the accuracy of the resulting model: with too few candidate solutions, the training algorithm did not have enough candidates to 'discriminate against.' Their results provide evidence that the size of the search space, including *both* high- and low-quality solutions, impacts pipeline performance.

[1] Accuracy is defined according to some evaluation metric, i.e., tag-sequence accuracy (Section 3.3.1, p. 40), F-score accuracy (Section 3.3.2, p. 41), etc.

6.1.3 Rank Accuracy

Section 6.1.1 discussed measuring the accuracy of the one-best candidate in a constraint set. However, while such a score may reliably indicate a high-quality one-best candidate, it may not provide a reliable ranking of the remaining candidates in the list. Thus in this section we discuss two metrics of *rank accuracy.* We can get an intuitive sense of the rank-order accuracy of our different data sets in Figure 6.1. In examining these graphs, we can see that the reranked output has a higher rank accuracy than the parser output; the reranked data-lines are closer to the ideal rank, shown as a dotted line in each graph. The difference between the 50-best ∪ CSLUt-constrained graph and the 50-best ∪ Reranker-constrained graph (bottom of the figure) is somewhat intriguing in that the CSLUt-constrained graph appears more scattered than the Reranker-constrained graph. In the next section we present a quantitative measure of rank-order accuracy to determine whether the visible difference between these graphs reflects a quantitative difference.

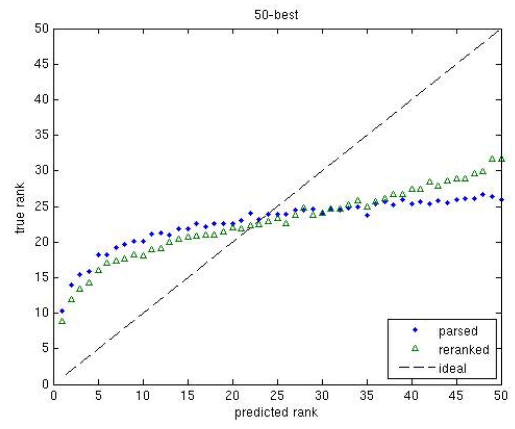
Margins

We could also measure rank-order accuracy by calculating a margin between the rank value assigned by any scoring metric and the true rank value of each candidate. Let i equal the true rank value of the candidate, as calculated by evaluating each candidate against the known true solution then ranking the candidates according to F-score, and j equal the rank of the candidate according to our scoring metric. Following Shen and Joshi [198], both *even* and *uneven margins* will be calculated. *Even margins* are calculated as $(i-j)$, such that ranking mistakes are equally penalized regardless of the rank value; thus ranking the second candidate as the third is as bad as ranking the 49th candidate as the 50th. *Uneven margins* are calculated as $(1/i - 1/j)$ so that errors in the lower ranks are not penalized as heavily as errors in the higher ranks. Table 6.3 presents the even and uneven margins calculated for each of our seven test conditions, where candidates are ranked either by the parser or the reranker. This table is essentially a numerical representation of the graphs in Figure 6.1. Note that perfect rankings (i.e., that match the dotted black lines shown in Figure 6.1) would have even and uneven margins of 0, and reverse-rankings (i.e., perpendicular to the dotted black lines in Figure 6.1) would have even margins equal to half the size of the ranked list $(n/2)$ and uneven margins equal to approximately $2e^{-n^{.25}}$.

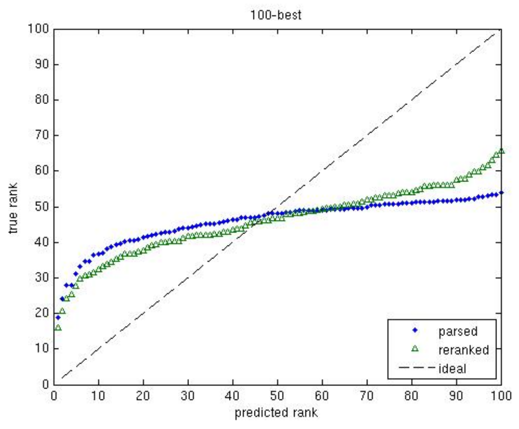
Table 6.3 confirms our intuition from Figure 6.1: the reranker-ordered candidates are closer to the true ranking than the parser-ordered candidates. The "improvement" in the uneven margins, measured as the difference between the parser-ordered uneven margins and the reranker-ordered uneven margins, is consistent across six of our seven test conditions; the Self-trained condition shows a smaller than expected improvement (given the size of its n-best lists). The improvement in the even margins is also slightly smaller than expected under the 50-best ∪ CSLUt condition. However, these differences are not deemed large enough to be of concern. We will return to the

		Parser-Ranked		**Reranker-Ranked**	
Condition	n	**Even Margin**	**Uneven Margin**	**Even Margin**	**Uneven Margin**
50-best	47.9	14.9	0.134	13.6	0.126
100-best	95.2	29.9	0.092	27.2	0.087
Self-trained	49.8	14.8	0.133	13.5	0.128
CSLUt-constrained	46.2	14.9	0.155	13.3	0.147
Reranker-constrained	46.9	14.7	0.152	13.3	0.144
50-best ∪ CSLUt	75.0	23.7	0.105	21.5	0.099
50-best ∪ Reranker	72.1	23.5	0.110	21.0	0.104

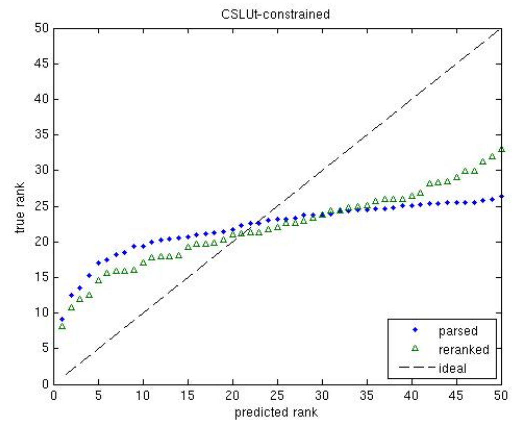
Table 6.3: Even and uneven margins calculated as a measure of rank-order accuracy.



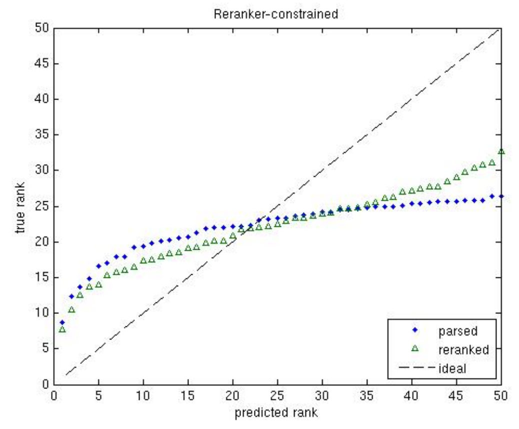
50-best



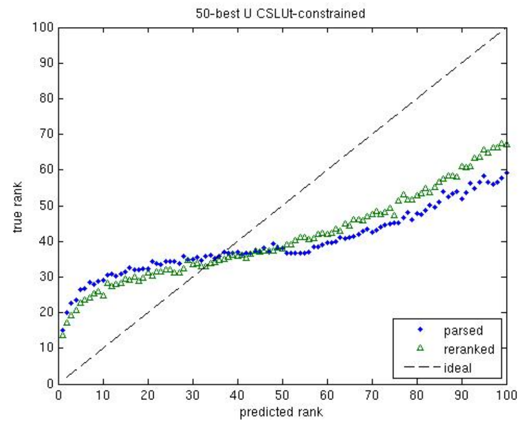
100-best



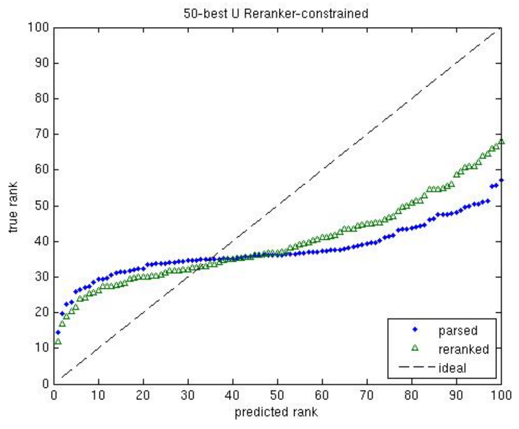
CSLUt-constrained



Reranker-constrained



50-best ∪ CSLUt-constrained



50-best ∪ Reranker-constrained

Figure 6.1: Rank-order accuracy of the parser probability score as compared to the rank-order accuracy of the reranker score. The true rank of each candidate is shown as a dotted black line.

Condition	n	Mean Reciprocal Rank		
		Reranker	Oracle	
		Best	Best	Worst
50-best	47.9	0.23	0.11	0.03
100-best	95.2	0.18	0.06	0.02
Self-trained	49.8	0.33	0.12	0.03
CSLUt-constrained	46.2	0.26	0.13	0.04
Reranker-constrained	46.9	0.37	0.14	0.04
50-best ∪ CSLUt	75.0	0.20	0.08	0.02
50-best ∪ Reranker	72.1	0.21	0.08	0.03

Table 6.4: Mean reciprocal rank of the oracle-best, oracle-worst, and reranker-best candidates.

notion of even and uneven margins in Chapter 7, when we introduce new metrics for ranking parse candidates in an n-best list.

Mean Reciprocal Rank

In this section we measure the *mean reciprocal rank* (MRR) as a measure of rank accuracy. Reciprocal rank is the multiplicative inverse of the rank of the correct answer; here we will define the correct answer to be either the reranker-best, oracle-best, or oracle-worst, candidate. Thus the mean reciprocal rank is the average of the reciprocal ranks of results over a set of n-best lists:

$$\text{MRR} = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{r_i} \tag{6.1}$$

where r_i is the rank of the correct answer. An MRR of 1.0 would mean that the correct answer is always ranked first. Table 6.4 shows the MRR of the reranker-best, oracle-best, and oracle-worst candidates in n-best lists ranked by parser-probability scores. We can see from the table that the reranker-selected candidate is usually fairly near to the top of the lists, with the oracle-best candidates slightly lower in rank. The oracle-worst candidates are consistently about halfway down the list.

6.2 Empirical Effects of Manipulating Accuracy

6.2.1 Artificially Inflated Oracles

In this section we experiment with artificially inflating oracle rates to determine the effects on downstream performance. For this set of experiments we simply appended the true reference into the n-best lists of parses input to a reranker stage. The reference candidates were added to (a) just the training set or (b) just the heldout set. We also experimented with adding the reference candidates to the training data but disallowing any features related to only the reference candidates, with the intuition that this would prevent the reranking model from over-weighting features that reliably indicated good solutions but which were unlikely to occur in the test set.

Note that the reference candidates were added by a brute force method, and thus we have no parser probability score for these constraints.[2] Therefore to create a baseline for comparison, we simply removed the parser-probability score from all of the n-best parses input to the reranker. Table 6.5 shows the results of these experiments, which demonstrate that artificially inflating the oracle of an n-best list proves detrimental to performance.

[2] We tried to constrain the parser to produce exactly the reference solution, similar to our parser-constrained experiments in Chapter 4, but experienced a high parse-failure rate (over 30%).

Condition	**Reranker-Best**
50-best	
“unweighted”	89.4
(a) reference in training	88.3
(b) reference in heldout	88.0
(c) reference, -reference features	88.4

Table 6.5: Results from artificially inflating oracle rates by inserting the reference candidate to (a) the training set or (b) the heldout set of n-best lists. Reference candidates were also added to the training set for (c), while features related only to the reference candidates were removed from the feature set.

6.3 Conclusion

We have shown in this chapter that the accuracy of a constrained space does not necessarily correspond to downstream performance. The literature of pipeline systems has not provided any explanations as to why one-best accuracy does not correspond to pipeline-final accuracy. In this dissertation we hypothesize that such discrepancies between intrinsic and extrinsic evaluation may be due to characteristics of the space that are not captured by accuracy metrics. For example, perhaps a downstream reranker would have benefited from a more diverse set of candidates or a smoother distribution of the probability scores over the candidates. Low-accuracy models have also been shown to be helpful, particularly in the case of voting pipelines [31, 108, 189, 191], and arguments have been made in the literature that the low-accuracy models help by increasing the *diversity* of the search space. Chapter 7 will be devoted to exploring the diversity and other characteristics of a constrained space to determine whether such characteristics better explain the effects on pipeline performance.

Chapter 7

Constraint Distribution

The objective of this chapter is to characterize a set of solution candidates[1] according to how the candidates are distributed throughout the defined space. Envision a set of candidates as data points scattered across a two-dimensional space.[2] We hypothesize that the distribution of these data points in the defined space (which is in fact defined *by* the set of data points) can affect downstream performance. For example, if the data points are scattered unevenly throughout the space, then a stochastic hill-climbing algorithm [157] in a downstream stage might find it difficult to step from point to point. Similarly, if the data points are far apart from each other, then it might be too costly to move from one point to the next, resulting in a sub-optimal solution because exploring the space was too expensive. On the other hand, if the data points are all closely clustered together then it might prove difficult to learn to discriminate amongst them.

Section 7.1 discusses several characteristics related to the distribution of candidates in a space. Section 7.2 introduces a number of metrics that we will use to formally capture these spatial characteristics, including F-score as a similarity metric, D-score as a related distance metric, Levenshtein edit distance applied to parse trees, graph-theory eccentricity to measure the diameter and area of a space, and k-nearest-neighbor metrics. In Section 7.3 we use these metrics to manipulate a search space in order to see exactly how changes in these spatial characteristics can affect pipeline performance.

Throughout this chapter we will be comparing pairs of candidates in a set. We assume the set of candidates to be ordered, and will examine some of the effects of changing the order within the set. There are three different types of pairwise comparisons that we will be conducting: *full pairwise*, *stepwise*, and *one-to-many* comparisons. In a full pairwise comparison, each candidate in the set is compared to every other candidate in the set; we typically take the average of the comparisons. In a stepwise comparison, we compare each candidate only to the next (or previous) candidate in the set, recalling that the set is assumed to be ordered. In a one-to-many comparison, a single candidate is chosen from the set, then every other candidate in the set is compared to the selected one. Figure 7.1 provides a visual representation of these three types of comparisons. In the figure, a constrained search space is represented as a graph, the candidates as nodes in the graph, and the comparisons between candidates as node connections in the graph.

[1] In this chapter we focus on fully-defined search spaces, and thus refer to input constraints as candidate sets. However, the metrics presented herein will generalize to other types of search space constraints, as discussed in Section 7.2.5.

[2] The solution space and therefore the search space of any NLP task is, of course, multi-dimensional, but we simplify for conceptual purposes here.

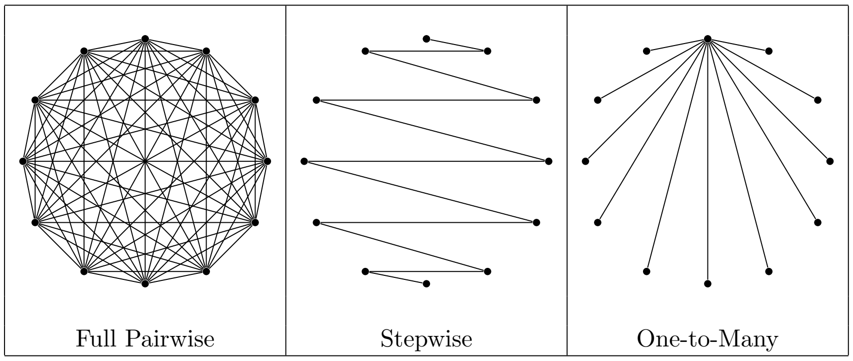


Figure 7.1: A visual representation of three different types of comparisons: (a) full pairwise, (b) stepwise, and (c) one-to-many.

7.1 Spatial Characteristics

There are a number of different spatial characteristics that we wish to examine, including *diversity, regularity, coverage*, and *density*. All of these contribute to an overall characterization of the space; one of these alone will not give a complete picture. We begin with the notion of *diversity*, which has often been argued to have a beneficial effect on downstream performance [108, 77, 167, 218]. We then make the counter-argument that diversifying a constrained search space does not necessarily lead to improved performance, and that the other spatial characteristics must also be taken into consideration. The effects of these spatial characteristics have been overlooked or ignored in the NLP literature.

7.1.1 Diversity

One of the few spatial characteristics that is often mentioned in the literature is that of candidate *diversity*, or diversity of the search space as defined by the candidates. We will define diversity in terms of the distance between each candidate in the space, where candidates that are further away from others in the space contribute more to the diversity of the space.[3] In terms of diversity in a pipeline, a more-diverse set of candidates will presumably correspond to a larger number of features for the downstream stage. However, we argue that simply increasing the number of features will not necessarily result in improved downstream performance, and that there are other factors at play which can also affect performance. These other factors include other characteristics of the distribution of candidates, as discussed in the remainder of this section.

As a simple example of diversity in a fully-specified search space, consider a 3-best list of POS-tag sequences to be reranked, shown in Figure 7.2. The first two sequences differ only by a single tag, and are thus very similar; the space defined just by these two sequences would not be very diverse. Now consider the third sequence, which differs from the first in four tags, or 66% of the sequence, and differs from the second sequence in three tags (50%). Adding this sequence to the search space increased the diversity of the space.

Another, more complex, example of diversity in a fully-specified search space is shown in Figure 7.3: a set of 50 parse trees for a single sentence, to be input to a reranker. In this figure, the difference between a pair of trees is visually represented: the red dashed lines indicate nodes that are present in the first tree but not the second of the pair, whereas the blue dotted lines indicate nodes present in the second tree but not the first.

[3] Note that here we are using a fairly vague, intuitive notion of diversity and distance; we will present formal metrics later in the chapter.

(1)	NN	NNS	VBD	IN	JJ	NN
(2)	NN	NNS	NN	IN	JJ	NN
(3)	VB	NNS	NN	IN	NN	VBG
	Stock	prices	rose	in	light	trading

Figure 7.2: 3-best list of POS-tag sequences input to a reranker, as an example of a discrete search space in which we examine both diversity and regularity characteristics.

To intuitively demonstrate the diversity in this space, we conducted a one-to-many comparison of each of the parse candidates against the first in the list, where the list is ordered by the score assigned to the parse by the baseline (upstream) parser. The first few candidates are fairly similar to the first, only differing by a few nodes in the tree. As we progress down the list (towards the lower right corner of the figure), we see larger differences between the parses later in the list and the first parse in the list. The differences contributed to the list by these parses increase the diversity of the space.

Sagae and Lavie [189] and Sagae and Tsujii [191] noted the positive impact of increased diversity for voted recombination. In a voted recombination stage, the partial solutions receiving the greatest number of "votes" in the input are combined to create the output solution. Sagae and Lavie [189] found that using several different parsers to generate sets of parse candidates for recombination resulted in improved performance as compared to using an n-best list of parses from a single parser to define the candidate set. They also noted that excluding the input from lower-performing parsers actually had a detrimental effect on the output of the recombination. The conclusion of both of these papers was that the diversity of a candidate set affects the output performance of voted recombination. In Chapter 4 (p. 72) we, too, found that recombining n-best lists from two different shallow-parse lists resulted in higher accuracy than the recombination of an n-best list from a single parser.

While diversity has proven beneficial in the context of a recombination stage, it may not necessarily perform as well in other contexts, such as a reranking stage. While we saw positive results for recombining lists of shallow parses in Chapter 4, we saw negative results when reranking the most-diverse set of full-parses (as discussed on p. 73). Unfortunately, diversity is *assumed* to be beneficial under the reranking paradigm due to its success under the recombination paradigm.

Despite the oft-referenced benefits of diversity, there has been no work to explicitly measure the diversity of a solution set. Previous work has relied on an *indirect* measure of the similarity of the candidates: the difference in candidates' accuracy. It would be more useful to have a quantitative metric of the diversity of a candidate set; Section 7.2 will present several such metrics. Additionally, "diverse" candidate sets are typically produced by using multiple different models to produce multiple solution sets, then combining sets. Section 7.3.1 presents a technique to systematically increase (or decrease) the diversity of a set, which will allow us to directly demonstrate how altering the diversity of the space affects pipeline performance.

7.1.2 Regularity

Another spatial characteristic is the *regularity* of the space. We will define regularity in terms of equidistance, i.e., whether or not the candidates in the space are equally-distant from its nearest neighbor(s). Note that regularity is a relative measure of distance, in that a space where the candidates are all far apart could have the same level of regularity as a space where the candidates are close together, provided that the distance separating the candidates is consistent within each space. The regularity of the search space can affect the search algorithm. A set of solution candidates with sparse coverage of the space may be easily separable; on the other hand, a set of candidates with dense coverage of the search space may allow for improved estimation of the

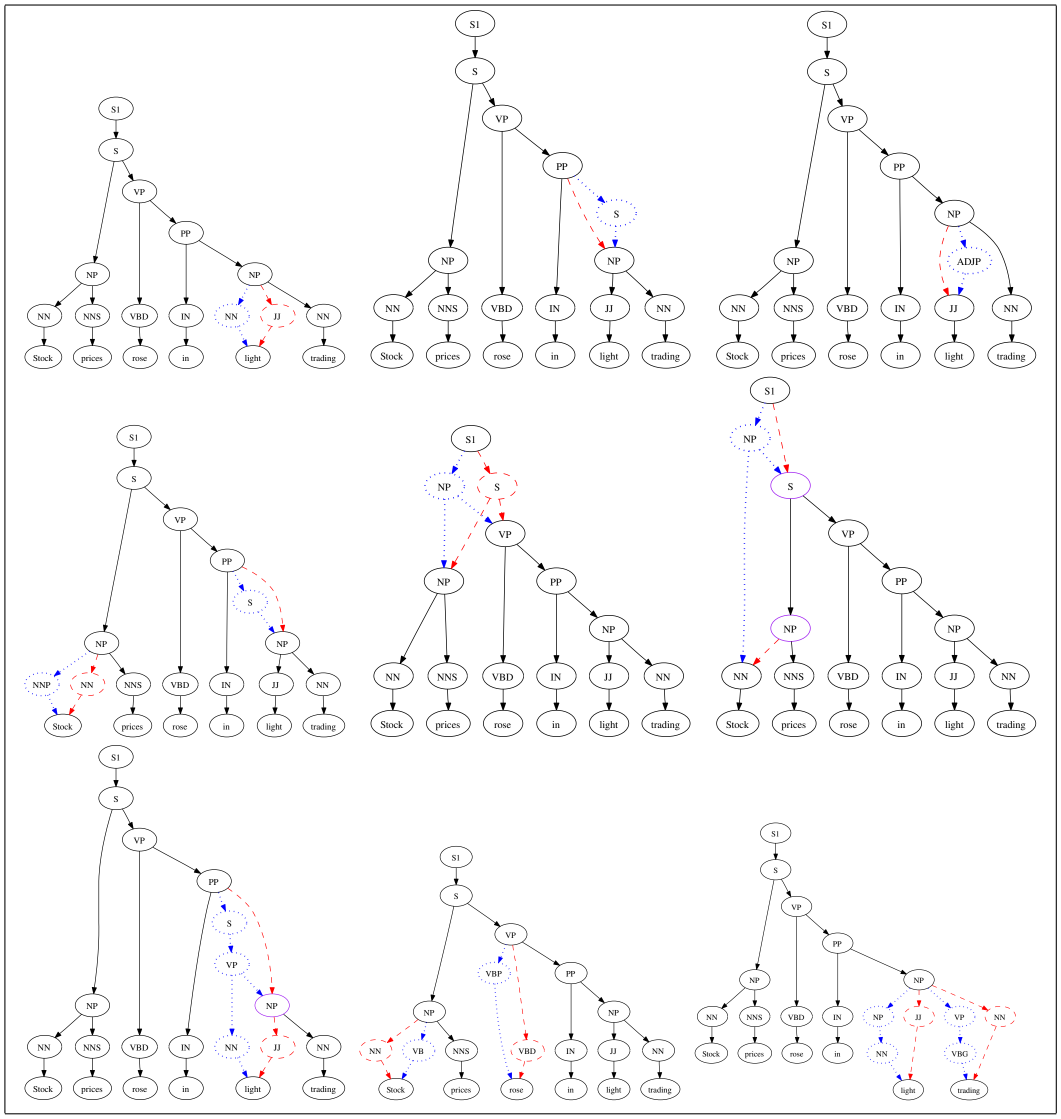


Figure 7.3: n-best list of parse trees input to a reranker, demonstrating the level of diversity in such a discrete search space.

search parameters.

As an example of regularity in a fully-specified search space, consider once again the POS-tag sequences in Figure 7.2 (p. 98). Since the first two sequences differ only by a single tag, they might be considered to be very close together in the space: only one "step" apart. In contrast, the third sequence differs from the first in four tags, or 4 steps, and differs from the second sequence in three tags (3 steps). Thus the step sizes in this space are irregular. Had every sequence differed by 3 tags or by 1 tag, then the space would be more regular in the sense that each step is the same size.

We can also discuss the regularity of the more complex parse search space represented in Figure 7.4. The parses in the list are ordered by the score assigned to the parse by the baseline (upstream) parser; if we conduct a stepwise comparison of these parses, then the first (parser-best) parse is compared to the second, the second is compared to the third, etc. This figure shows that the differences from one parse tree to the next vary quite a bit; the first few pairs are only different in one or two nodes, but near the end of the list (lower right corner of the figure), the differences

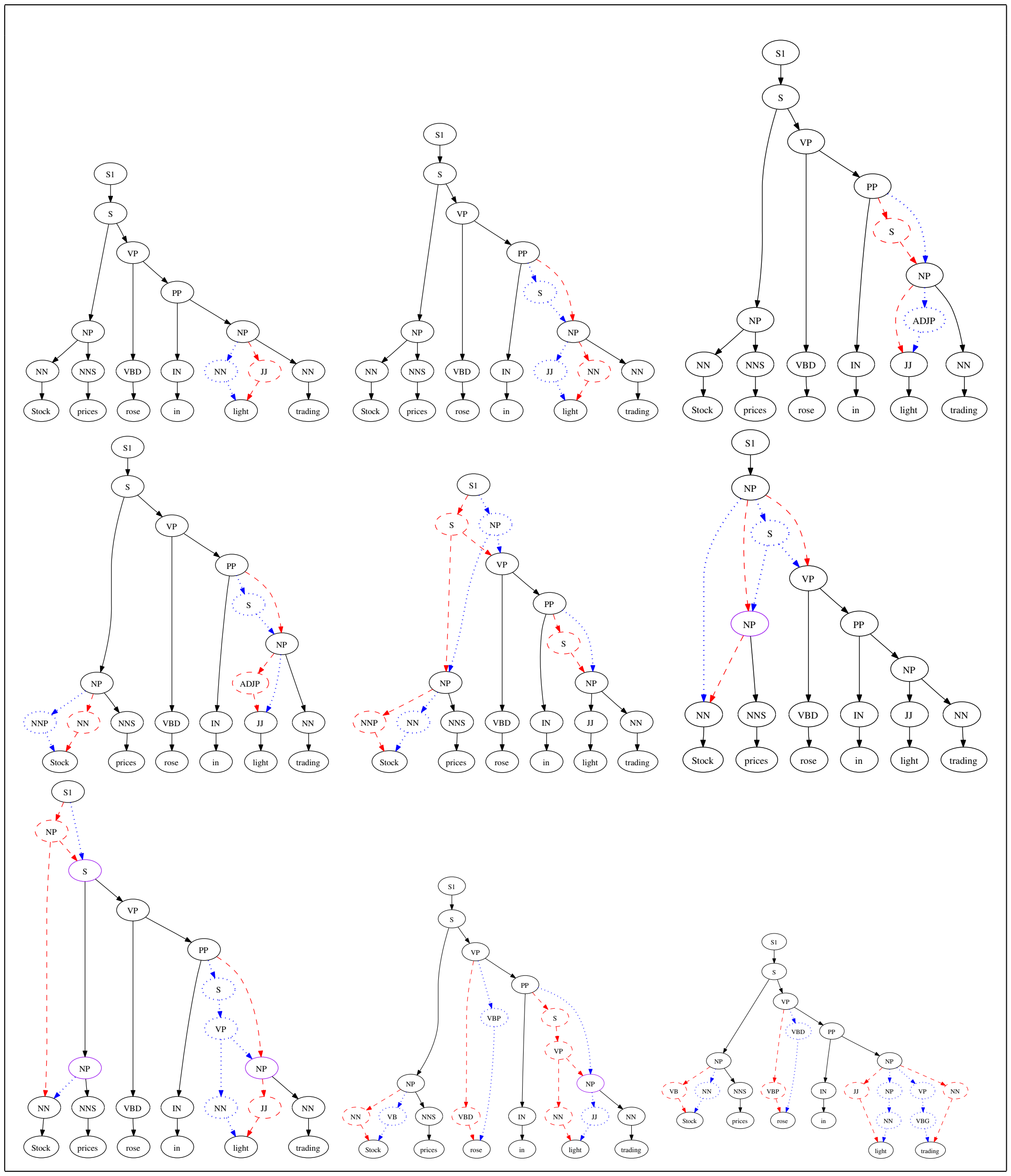


Figure 7.4: *n*-best list of parse trees input to a reranker, demonstrating the level of regularity in such a discrete search space.

between parses become a bit larger. Thus, had we cut off the list of parses at the 3-best rather than 10-best, the space would have been more regular; adding more parses decreased the regularity of the space while increasing its diversity.

7.1.3 Density

Another spatial characteristic, which is closely related to diversity and regularity, is that of *density*. We will define density as a measure of how "close" candidates are in a space; a space consisting of candidates that are very close together will be very dense, whereas a sparse space would consist of candidates that are spaced far apart from each other. A space constrained by the first three candidates in Figure 7.4 would be more dense than the space represented by the last three candidates in the figure, since the pairwise distance between the first three candidates is smaller than the pairwise distance between the last three candidates.

If the constrained search space is very dense, with the candidates tightly clustered together, then the downstream stage must learn to select from a space wherein the features of the solution candidates only vary along a few dimensions. If, on the other hand, the candidates are more spaced out and thus the constrained area is more sparsely populated, then the solution candidates may be more easily separable into "good" solutions and "bad" ones. Thus, despite the fact that widely-spaced candidates in a sparse search space correspond to a larger area with more features to consider, the solution selected from such a subset may be better than one selected from a smaller, denser set of solutions.

7.1.4 Coverage

The fourth characteristic to be explored in this chapter is that of *coverage*. We intuitively define coverage in terms of "gaps" in the space. Imagine again that the candidates are discrete points in a two-dimensional space; if the candidates provided perfect coverage of the space, then we could start at any one of those points and, taking a single step in any direction (without leaving the defined space), reach another candidate in the space. The "single step" would be defined according to the task at hand: in tagging, that single step would be changing a single tag in the sequence, whereas in parsing it might be changing a single node in the parse tree, and in machine translation it could be changing the translation of one of the words, or changing the position of one word in the sentence. Of course, our definition of the coverage of a space only applies to discrete spaces—a continuous space would, by definition, have complete coverage—but an NLP search space is discrete so the notion of coverage is a valid one for this research. Incomplete coverage might affect a downstream search algorithm in two ways: by creating an uneven space, where moving from point to point in the space is difficult due to uneven step-sizes between gapped points; and by forcing the algorithm to choose between two imperfect candidates because the candidate that would have been preferred by the algorithm is missing, or gapped.

As an example of incomplete coverage, consider once more the POS-tagging search space represented in Figure 7.2 (p. 98). Note that since there are three differences between the second and third sequence, there is a gap between these two in terms of coverage of the space. Furthermore, there are four differences between the first and third sequence, only two of which are covered by the second sequence. To cover the gap between the second and the third sequence, we will need an additional five sequences. To cover the gap between the first and the third sequence, we would need fourteen additional sequences (minus one which is already represented by the second sequence). If we generated all of these sequences we would have the full-coverage space shown in Figure 7.5, with the additional sequences listed as candidates 4–22. Inadequate coverage of the solution space can be problematic: in the example provided, the first sequence correctly identified "rose" as a verb but incorrectly identified "trading" as a noun, while the third sequence correctly identified "trading" as a verb but incorrectly identified "rose" as a noun. Given just these three sequences as the search space, a search algorithm would be forced to choose between two sub-par solutions, since the correct solution in its entirety is missing from the space even though the correct solution is represented piecewise in the space.

7.2 Distance and Similarity Metrics

In the previous section of this chapter, we described the diversity, regularity, coverage, and density of a search space in intuitive terms, loosely referring to the distances between candidates in the

(1)	NN	NNS	VBD	IN	JJ	NN
(2)	NN	NNS	NN	IN	JJ	NN
(3)	VB	NNS	NN	IN	NN	VBG
from sequence (1) to (3):						
(4)	VB	NNS	VBD	IN	JJ	NN
(5)	NN	NNS	VBD	IN	NN	NN
(6)	NN	NNS	VBD	IN	JJ	VBG
(7)	VB	NNS	NN	IN	JJ	NN
(8)	VB	NNS	VBD	IN	NN	NN
(9)	VB	NNS	VBD	IN	JJ	VBG
(10)	NN	NNS	NN	IN	NN	NN
(11)	NN	NNS	NN	IN	JJ	VBG
(12)	NN	NNS	VBD	IN	NN	VBG
(13)	VB	NNS	NN	IN	NN	NN
(14)	VB	NNS	NN	IN	JJ	VBG
(15)	VB	NNS	VBD	IN	NN	VBG
(16)	NN	NNS	NN	IN	NN	VBG
from sequence (2) to (3):						
(17)	VB	NNS	NN	IN	JJ	NN
(18)	NN	NNS	NN	IN	NN	NN
(19)	NN	NNS	NN	IN	JJ	VBG
(20)	VB	NNS	NN	IN	NN	NN
(21)	NN	NNS	NN	IN	NN	VBG
	Stock	prices	rose	in	light	trading

Figure 7.5: The POS-tag sequences required to completely cover the space represented by the 3-best list in Figure 7.2.

space. In this section, we will formally define several different metrics for comparing candidates in a space, and discuss the requirements of a distance metric, as well as the pros and cons of using these metrics to characterize a space.

7.2.1 F-score and D-score

Context-free parse candidates are evaluated according to the F_1 metric, as discussed in Section 3.3.2 (p. 41). We can use the F-score to calculate the pairwise similarity of two parse candidates; two highly-similar parses will have a high pairwise F-score near 1.0, versus two very different parses which will have a much lower pairwise F-score.

F-score provides a *similarity* metric, indicating a level of shared information between two parse trees. It would be convenient if one could straightforwardly use $1-\mathrm{F}$ to calculate the *distance* between two trees. Unfortunately, such a calculation is invalid as a distance metric because it does not meet the *triangle inequality* condition [143], where the length of the path between two points must be less than or equal to the path created by adding an intermediary point:

$$\triangle AC \leq \triangle AB + \triangle BC. \tag{7.1}$$

To see that this condition is not met with the $1-\mathrm{F}$ metric, consider the case where $A=\{x, y\}$, $B=\{x, y, z\}$, and $C=\{y, z\}$. Using $1-\mathrm{F}$ to calculate the distance ($\triangle$) between the three sets gives us $\triangle AC=1-\frac{1}{2}$, $\triangle AB=1-\frac{4}{5}$, and $\triangle BC=1-\frac{4}{5}$. Since $\frac{1}{2}>\frac{1}{5}+\frac{1}{5}$, the "distance" calculated by $1-\mathrm{F}$ violates the necessary condition of triangle inequality.

Fortunately, Yianilos [216] demonstrated that a metric meeting each of the conditions necessary for a true distance metric, including the triangle inequality condition, can be derived from the same calculations as used for similarity metrics such as F-score. (See that paper for a proof in the general

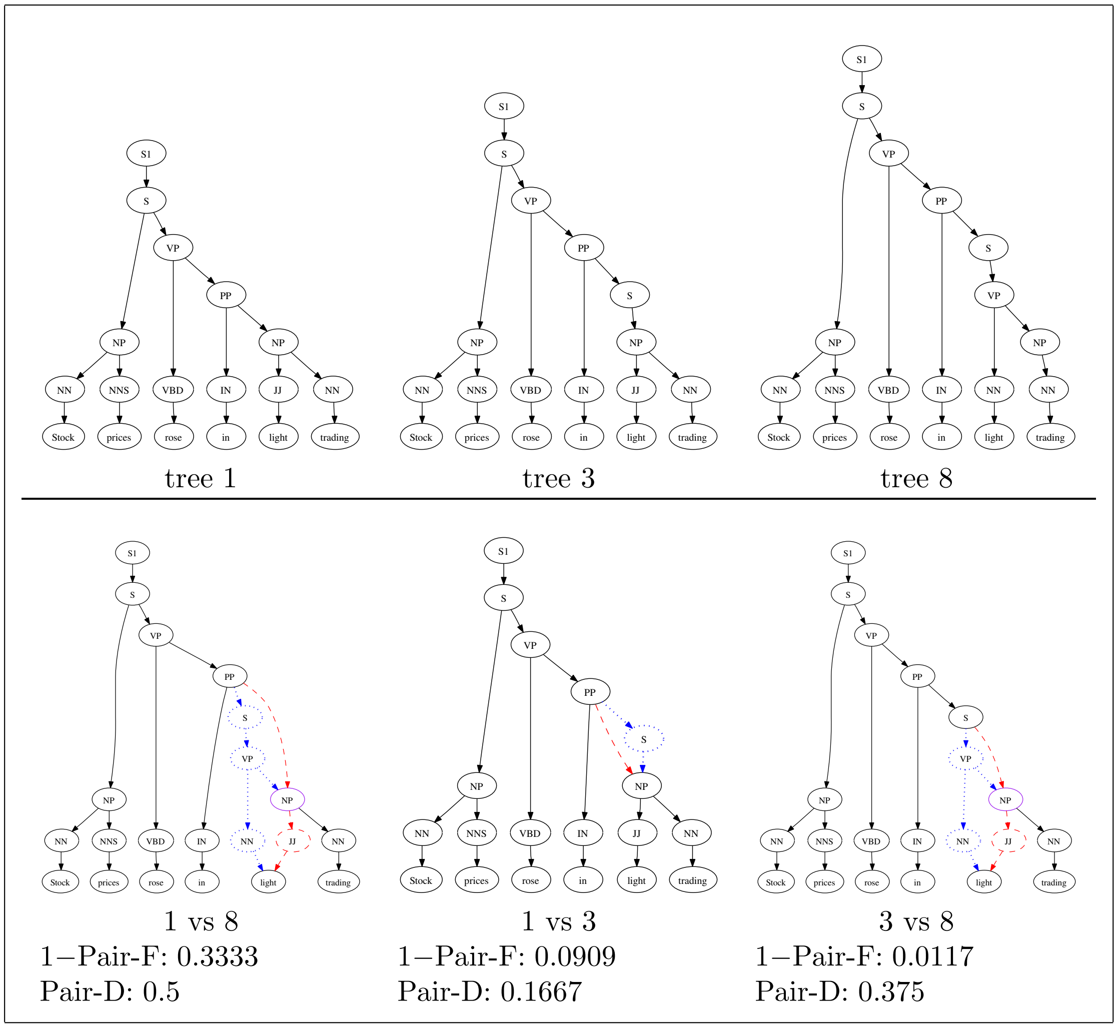


Figure 7.6: Three trees, each compared to each of the others using the F-score similarity metric and the D-score distance metric. Using 1−F to approximate distance violates the triangle inequality condition as shown.

case.) This metric shall be termed the D-score, for *distance metric*, which is defined as follows:

$$D = 1 - \frac{|A \cap B|}{|A \cup B|}. \tag{7.2}$$

Note that since $|A \cup B|$=$|A| + |B| - |A \cap B|$, calculating this distance metric requires just the same measurements as calculating the F-score metric. Thus we can calculate pairwise F-scores, to determine the average similarity amongst candidates in an n-best list, as well as pairwise D-scores, to determine the average distances between candidates in the list.

Figure 7.6 shows three parses for our example sentence, where using 1−F as a distance metric violates the triangle inequality and using the D-score as the distance metric does not. The 1−F "distance" between trees 1 and 8 in Figure 7.6 (as approximated by calculating the pairwise F-score, then subtracting that score from 1), is larger than the sum of the 1−F distances from tree 1 to 3 and from tree 3 to 8, thus violating the triangle inequality condition. We can measure how frequently the triangle inequality is violated by the 1−F distance score, to determine whether the violation only happens in pathological cases or is much more common. In a full pairwise comparison of the 50-best parses output by the Charniak parser for WSJ section 24, about 4% of the pairwise comparisons violate the triangle inequality condition. This condition is not violated, of course, when using the D-score metric (Equation 7.2) to perform the pairwise comparisons.

To calculate the similarity of and distance between parse candidates in an n-best list, we compare each candidate in the list to each of the other candidates in the list and take the averaged pairwise score for each candidate; we will refer to these scores as *Pairwise-F* and *Pairwise-D* scores. We could use these pairwise metrics simply to re-order an n-best list of parses. Table 7.1 shows the results of ordering n-best parses, as generated under the seven different conditions we are investigating, according to each parse candidate's parser score, pairwise F-score, and pairwise D-score. Here we are simply exploring whether re-ordering the lists according to similarity and distance scores, rather than baseline system scores, will change the top-ranked candidate. Indeed, we see in the table that, under some of the test conditions, both of the pairwise metrics do select a better top candidate than the baseline system score did. The F-score of the top-ranked candidate is significantly improved (paired t-test, $p<0.001$) under the 50-best, 100-best, and 50-best ∪ Reranker-constrained conditions (bolded and marked by an asterisk in the table) using either of the pairwise scores to order the candidates. We also see a small but statistically insignificant improvement under each of the other conditions as well.

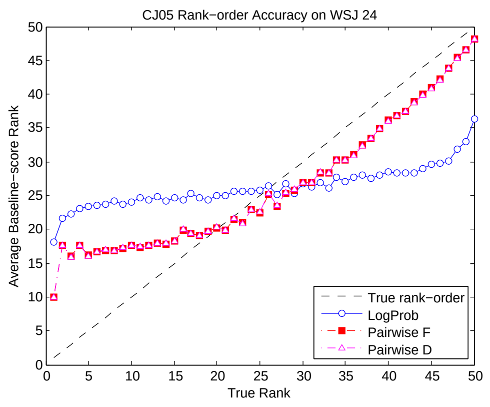
In Table 7.1 there seems to be little difference between the results of using the Pairwise-F and Pairwise-D metrics to re-order the candidates, with just a small difference shown in the table for the Reranker-constrained and the 50-best ∪ CSLUt-constrained conditions. Even though the F-score of each set is nearly identical, that does not necessarily mean that each pairwise score is selecting the same candidate as the one-best candidate. Furthermore, since we created the D-score to meet the formal conditions required of a distance metric without much regard to how well it correlated to the evaluation metric (F-score), it is a fortunate coincidence that selecting the candidate *closest* to the other candidates in the space, rather than the candidate *most similar to* the others in the space, results in similar accuracy rates.

Note the similarity between these methods of pairwise comparison and minimum Bayes–risk inference [106] (see also [97, 133, 135, 195]). The value resulting from the full pairwise comparison represents the expected loss of the hypothesized candidate with respect to the remainder of the n-best list. In this chapter we use the F-score and D-score metrics to characterize the diversity of candidates within a space. In Chapter 8 we will explore how these metrics behave as weighted distributions over the space, and whether they provide utility as weight-distribution features in downstream stages.

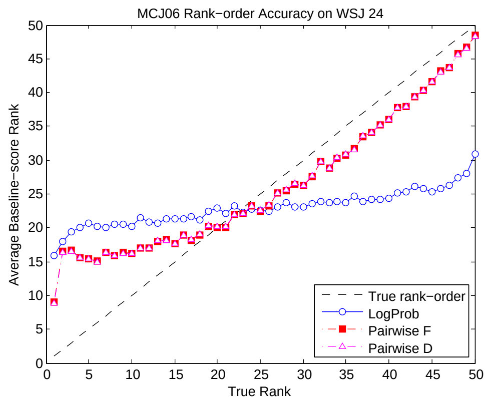
As shown in Table 7.1, re-scoring a set of candidates may effect a total re-ordering of the list. Figure 7.7 shows the rank-accuracy of the Pairwise-F and Pairwise-D scores, as compared to the

Condition	**Parser Score**	**Pairwise F-score**	**Pairwise D-score**	
50-best	88.9	**89.4**	**89.4**	*
100-best	88.8	**89.3**	**89.3**	*
Self-trained	90.2	90.4	90.4	
CSLUt-constrained	88.4	88.6	88.6	
Reranker-constrained	89.6	89.7	89.8	
50-best ∪ CSLUt	89.4	89.4	89.5	
50-best ∪ Reranker	89.2	**89.9**	**89.9**	*

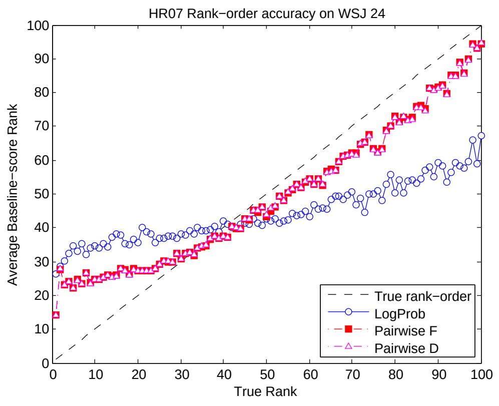
Table 7.1: F-scores on WSJ section 24 of 50-best output parses where candidates were re-scored and re-ordered by pairwise F-score or D-score comparison values. Bolded results and asterisks indicate conditions where the re-ordering resulted in a statistically-significant (paired t-test, $p<0.01$) improvement in F-score accuracy of the top-ranked parse candidate.



50-best



Self-trained



50 ∪ CSLUt

Figure 7.7: Rank-order accuracy produced by re-ordering the n-best candidates according to the log-probability score, Pairwise-F score, and Pairwise-D score of each candidate; the Pairwise-F and Pairwise-D lines appear to overlap in these graphs. Compares candidate rank, as determined by one of the three baseline-scores, against the true rank of each candidate (dotted black line).

parser probability score. As can be inferred from the graphs and as shown in Table 7.2, both of the pairwise metrics come closer to the true ranking (shown as a dotted line along the diagonal), and thus more closely approximate the true candidate ranking than does the log-probability score. Recall from Chapter 6 that a perfect ranking would result in even and uneven margins of 0. From the table we see that the pairwise metrics provide a more accurate ranking than the parser log-probability score. More specifically, we can see from Figure 7.7 that the pairwise metrics provide a more accurate ranking for the bottom half of the n-best lists, which is reflected in Table 7.2: the pairwise metrics show a great improvement in the even margins over the parser-ranking, whereas the improvement is much smaller for the uneven margins (where lower-ranked candidates are not weighted as heavily as top-ranked candidates). Note that in the Self-trained condition, the pairwise-ordering results in an uneven margin smaller than the 50-best condition, which is somewhat surprising given that the lists are of similar size. Interestingly, the pairwise-ordered lists have smaller even margins than the reranker-ordered lists under all of the test conditions (see Table 6.3, p. 92), and smaller uneven margins under the Self-trained, CSLUt-constrained, and Reranker-constrained conditions. We saw this improvement in rank-order translate to an improved one-best accuracy using the pairwise metrics (refer back to Table 7.1); unfortunately the pairwise

		Parser-Ranked		**Pairwise*-Ranked**	
		Even	**Uneven**	**Even**	**Uneven**
Condition	n	**Margin**	**Margin**	**Margin**	**Margin**
50-best	47.9	14.9	0.134	9.4	0.126
100-best	95.2	29.9	0.092	18.3	0.090
Self-trained	49.8	14.8	0.133	8.7	0.118
CSLUt-constrained	46.2	14.9	0.155	9.1	0.139
Reranker-constrained	46.9	14.7	0.152	8.9	0.131
50-best ∪ CSLUt	75.0	23.7	0.105	14.8	0.100
50-best ∪ Reranker	72.1	23.5	0.110	13.6	0.103

Table 7.2: Even and uneven margins calculated as a measure of rank-order accuracy, for parser-ranked n-best lists as compared to rankings produced by the pairwise metrics.
*Pairwise-F and Pairwise-D scores resulted in similar margins; only Pairwise-D is shown here.

metrics did not prove beneficial as reranking features, as will be shown in Chapter 8.

7.2.2 Levenshtein Distance

Both the F-score and D-score metrics produce length-normalized scores, from [0,1] inclusive. Such normalization accounts for size differences in the comparison objects (i.e., shorter versus longer word sequences, or smaller versus larger trees), but does not provide an absolute distance between two objects. For example, in Figure 7.6 we saw that the D-score distance between trees 1 and 8 is 0.5. However, looking at the two trees and their difference-tree shown in the lower-left corner of the figure, we can see that there are four nodes (including POS-tag nodes) that differ between the two trees: (S,4,6), (VP,4,6), (NN,5,5), and (JJ,5,5). A more intuitive distance-metric, then, might be to simply count the number of nodes that differ between a pair of trees. For such a metric, we can use the Levenshtein distance metric [142].

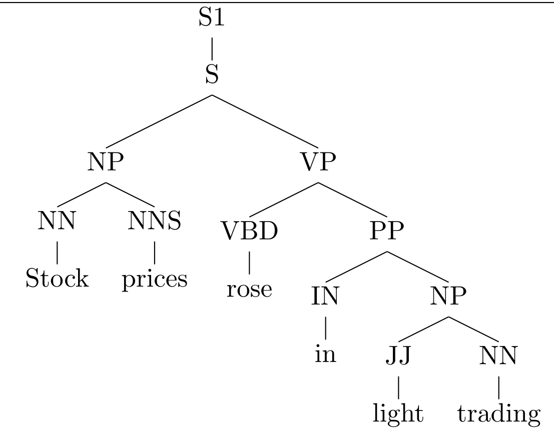
Levenshtein distance is calculated as the minimal number of edits required to transform the source sequence into the target sequence. There are three allowable types of edits: insertion, deletion, and substitution. All edit operations are assumed to be made on a single entity in the sequence, i.e., a single character in a word, or a single word in a sentence. For example, to transform the word "trading" into the word "raining," the following sequence of operations would be performed:

1. delete 't' (trading → rading)
2. substitute 'n' for 'd' (rading → raning)
3. insert 'i' (raning → raining).

As we can see, "trading" was transformed into "raining" in three steps, i.e., three edits. Of course there are other ways to perform the transform, some of which require the same number of steps; however, there is no transform that requires *fewer* than three edits, thus three is the Levenshtein distance between "trading" and "raining."

Levenshtein distance is frequently used to calculate the difference between strings, but we can also calculate the Levenshtein distance between a pair of parse trees. By representing a parse tree as an ordered set of nodes (ordered by start index and span), we can perform the same distance calculation on two ordered sets of nodes as on a sequence of characters. In fact, as discussed in Section 3.3.2 (p. 41), the commonly-used scoring script `evalb` represents parse trees as a set of labeled nodes. Figure 7.8 shows a parse tree and the sequence of labeled nodes to represent the tree.

(a)



(b) [(S1,0,6), (S,0,6), (NP,0,2), (NN,0,1), (NNS,1,2), (VP,2,6), (VBD,2,3), (PP,3,6), (IN,3,4), (NP,4,6), (JJ,4,5), (NN,5,6)]

Figure 7.8: Context-free tree (a) and its representation as a set of labeled brackets (b).

Thus if we wish to compare trees 1 and 8 from Figure 7.6 (p. 103), tree 1 would be represented as follows:

[(S1,0,5), (S,0,5), (NP,0,1), (NN,0,0), (NNS,1,1), (VP,2,5), (VBD,2,2), (PP,3,5), (IN,3,3), (NP,4,5), (JJ,4,4), (NN,5,5)]

and tree 8 from the same figure:

[(S1,0,5), (S,0,5), (NP,0,1), (NN,0,0), (NNS,1,1), (VP,2,5), (VBD,2,2), (PP,3,5), (IN,3,3), (S,4,5), (VP,4,5), (NN,4,4), (NP,5,5), (NN,5,5)].

If we treat each node in the set as a character in a string, then we can calculate the Levenshtein distance between the two sets—thus calculating the Levenshtein distance between the two trees. The naïve method to transform the first set into the second would be to simply delete any nodes in the first set that are not in the intersection of the two sets, and insert any nodes in the second set that are not in the intersection, which would result in six edits. There are actually only four edits required to transform tree 1 into tree 8:

1. substitute (NP,5,5) for (NP,4,5)
2. substitute (NN,4,4) for (JJ,4,4)
3. insert (S,4,5)
4. insert (VP,4,5),

resulting in a Levenshtein distance of four. Looking at the difference-tree of tree 1 versus tree 8 in the lower-left corner of Figure 7.6, this value makes some intuitive sense. Indeed, we can see that tree 8 has two more nodes than tree 1 (two insertions), and two nodes that differ only by span or label[4] (two substitutions).

The Levenshtein distance is commonly computed using a bottom-up dynamic programming algorithm which is $O(m * n)$ in time and space. There are several simple upper and lower bounds on Levenshtein distance: it is always at least the difference of the sizes of the two strings ($|m - n|$);

[4] Substitutions where the node-label differs are difficult to show in a difference-tree, and are thus visually represented as an insertion and a deletion.

it is at most the length of the longer string; it is zero if and only if the strings are identical; and if the strings are the same size ($m == n$), then the Hamming [105] distance[5] is an upper bound on the Levenshtein distance.

Note that using the Levenshtein distance instead of some other distance metric, like D-score, will not change the relative ranking of any set of objects from the ranking defined by the other distance metric. The added value of the Levenshtein distance is that it provides an intuitive measure of the absolute distance between two objects rather than the relative 'distance.' Thus we will use the Levenshtein distance metric extensively throughout the next few sections to give a sense of the distribution of the candidates in a pipeline search space.

7.2.3 Metrics from Graph Theory

It would be a simple matter to represent a candidate-defined search space as a fully-connected graph, where each node represents one of the candidates. Thus, in this section we will explore a number of different metrics from graph theory to measure spatial characteristics.

Eccentricity and Diversity

In graph theory, the maximum distance from one node to any other in the graph is called the *eccentricity* of that node [32]. The largest eccentricity value over all the nodes in the graph is called the *diameter* of the graph; the *radius* of the graph is defined as the minimum eccentricity value of all the nodes in the graph. If we assume that a fully-defined search space represented by a set of solution candidates is a fully-connected graph, with each candidate representing a node in the graph, then we can calculate the eccentricity of each candidate by conducting a full pairwise comparison with one of our distance metrics (Levenshtein edit distance or D-score).

Table 7.3 shows the average calculated diameter and radius (averaged maximum eccentricity and averaged minimum eccentricity, respectively) for each of our seven conditions. We used the Levenshtein distance metric to compare parse candidates, allowing only for deletions and insertions (no substitutions), and including part-of-speech tags in the comparison. We can use the diameter values as an estimate of the *size* of the space, rather than using the number of candidates in the list to represent size. In fact, the diameter of a space and the number of candidates in the space seem to have a proportional relationship in the table: the larger candidate sets (100-best, 50-best$\cup$CSLUt-constrained, and 50-best$\cup$Reranker-constrained) are also larger in diameter. However, the relationship is not linear; even though the 100-best set has nearly twice as many candidates as the 50-best set, its diameter is much less than twice the diameter of the 50-best space.

Note that in graph theory, diameter $\leq 2*$radius rather than the more expected relation of diameter $= 2*$radius, intuitively because there is no guarantee that there will be a node at the exact center of the graph. Indeed, the graph-theory relation holds true here, as expected, indicating that there is not always a parse candidate occupying the exact center of the space. We will return to this notion of central and peripheral nodes in Section 7.2.4 and again in Section 7.3.1.

We measure the density of the space by calculating the *area* of the space as radius$^2\pi$, then dividing the number of candidates in the space by this area.[6] We could also take the inverse of this measurement to give an estimate of the area occupied by one candidate. The rightmost columns of Table 7.3 show the density and the average area per candidate for each of our seven conditions. In comparing the 50-best condition to the CSLUt-constrained and Reranker-constrained conditions, we see that the constrained spaces are slightly more dense. This is unsurprising, given what we

[5] Calculated as the number of substitutions required to change one string into another where each string is the same length.

[6] Population density is taken as a measurement of population per unit area (or unit volume).

Condition	n	**Avg Diameter**	**Avg Radius**	**Density (n/Area)**	**Density^{-1} (Area/n)**
50-best	49.9	17.7	11.0	0.19	8.5
100-best	99.5	20.3	12.5	0.28	5.5
Self-trained	49.8	18.7	11.6	0.17	9.5
CSLUt-constrained	49.0	16.1	10.3	0.23	7.7
Reranker-constrained	48.8	16.0	10.1	0.23	7.4
50-best ∪ CSLUt	74.9	19.8	12.3	0.23	7.2
50-best ∪ Reranker	72.1	19.1	11.7	0.23	6.8

Table 7.3: Diameter-distance and radius-distance of parse candidates produced under various conditions; distance is measured as Levenshtein distance between sets of parse nodes.

know about the conditions under which these lists were produced: we expected a constrained space to be more densely populated than an unconstrained (or less-constrained) space. Surprisingly, the union of these lists (50-best ∪ CSLUt and 50-best ∪ Reranker) results in the same density as either of the constrained lists on their own, though denser than the 50-best list on its own.

Somewhat surprisingly, the 100-best space is quite a bit more dense than the 50-best space. This is probably related to the manner in which the Charniak [41] parser fills a parse-chart to produce n-best parses, and will be examined more closely in Section 7.3.3. Also surprisingly, the space generated under the Self-trained condition, which we expected should correspond fairly closely to the 50-best condition, is just slightly more sparse than the 50-best space. Unlike the base-constrained conditions, there is nothing inherent in the manner by which the Self-trained space is produced that would explain this discrepancy in density. Does it hint that the reranking paradigm does not perform as well on sparse spaces as compared to dense spaces? In Section 7.3 we will directly manipulate the density of a space in order to explore this possibility.

Path Distance and Regularity

We again borrow from graph theory, utilizing the definition of a *path*[7] to help measure the regularity and coverage in a space. In graph theory, a path is an alternating sequence of nodes and edges that starts and ends with a node, where the starting and ending nodes of the path are distinct. A *Hamiltonian path* [104] is a path which passes through every node in the graph once and only once. A node-edge-node sequence in a path is only permissible if the nodes are connected in the underlying graph. In graph theory the *length* of a path is calculated as n-1, where n is the number of nodes visited; if a node is visited more than once, it is counted each time it is visited. In this dissertation we will also define the *distance* of a path as the sum of the distances between each node along the path, where distance between nodes is calculated using one of our distance metrics (Levenshtein edit distance or D-score).

Since we typically have an arbitrary ordering of the candidates in a fully-specified space (ordered by the score of the candidate from the upstream stage), we can create a path that follows this ordering. By examining the characteristics of such a path, we can characterize the ordering provided by the upstream stage. Furthermore, since candidate-ordering is often used (either implicitly or explicitly) as a feature in downstream stages, in Chapter 8 we will use this path-distance metric again in order to explore how ordering characteristics may correspond to downstream performance in the pipeline. We can also re-order the candidates according to their similarity to or distance from each other, as was done in Table 7.1, and create a path that follows that ordering. Each of these paths, following either the baseline-parser score, the F-score similarity metric, or the D-score distance metric ordering, is a Hamiltonian path by definition, since the path will pass through

[7] Also referred to as an *open walk*.

		Path-Distance to n		
Condition	n	**Base**	**Pair-F**	**Pair-D**
50-best	47.9	355	235	233
100-best	95.2	803	502	499
Self-trained	49.8	375	252	249
CSLUt-constrained	46.2	332	213	211
Reranker-constrained	46.9	328	214	211
50-best ∪ CSLUt	75.0	557	397	394
50-best ∪ Reranker	72.1	531	365	363

Table 7.4: Average path-distance from the parser-best candidate to the parser-worst candidate, through the space of all parse candidates produced under various conditions; distance is measured as Levenshtein distance between sets of parse nodes. Paths follow the candidate-ordering as defined by the baseline parser-score (Base), pairwise-F score (Pair-F), or pairwise-D score (Pair-D).

each node (candidate) exactly once. Since each of these paths will pass through every candidate in the defined space, the same nodes will be covered by each path and the length of each path will be identical. However, the distance of each path, measured as the sum of the distance (or difference) between each node, may differ. Table 7.4 shows the average path distance using these three ordering methods under our seven conditions under investigation.

By comparing the Base Path-Distances (second column) of Table 7.4 to the values in the rest of the columns in the table, we can clearly see that path-distance is related to the number of candidates in a space. The path-distance is greatest under the 100-best condition, which also has the greatest number of candidates; and the path-distance is smallest under the Constrained conditions, which also have the fewest number of candidates. These results indicate that the path through an entire set of candidates must travel a greater distance when following the order defined by the baseline parser-score than when following the order defined by either the pairwise F-score similarity metric or the pairwise D-score distance metric. This tells us that the parser score does not tend to order candidates by similarity (which we will see again in Chapter 8, p. 126), and instead of generating an efficient path through the space, forces the path to jump from candidate to candidate without regard to their location in the space defined by the candidate set. The Pair-F and Pair-D values differ very little from each other, telling us once again that there is very little difference between the total path-distances of the candidates ordered by the Pairwise-F similarity and the candidates ordered by the Pairwise-D distance.

All three of the paths represented by Table 7.4 measure the distance required to cover the entire space. However, it may not be necessary to walk the entire space. In Chapter 6 (p. 91) we discussed the notion of oracle-best candidates. One could imagine that the only path necessary to walk within a space is that from some starting point (the baseline-best candidate, for instance) to the oracle candidate. We can calculate the distance of such a path, as shown in Table 7.5. In this table, as opposed to Table 7.4, there is very little difference between the path-distances in a baseline-parser ordered space versus a similarity- or distance-metric ordered space. However, the *length* of the path, calculated as the number of nodes in a path and shown in the rightmost columns of Table 7.5, is much longer for the similarity- and distance-ordered candidates than for the baseline parser-ordered candidates, at nearly double the length under all conditions. The bolded numbers in the table indicate that the path-distance was noticeably greater for the similarity- and distance-ordered candidates under the 50-best ∪ CSLUt-constrained condition. Does the difference between this condition and the others indicate that path-distance to the oracle candidate can influence downstream pipeline performance? We will examine the effects of path-distance more closely in Section 7.3.2.

Given the nearly-identical values for the path distances, it is perhaps surprising that the path

Condition	n	**Path-Dist to Oracle** Base	Pair-F	Pair-D	**Path-Length to Oracle** Base	Pair-F	Pair-D
50-best	47.9	63	60	60	9.5	15.0	15.0
100-best	95.2	120	119	119	15.6	27.7	27.6
Self-trained	49.8	56	55	55	8.4	13.7	13.6
CSLUt-constrained	46.2	49	51	51	8.0	13.7	13.7
Reranker-constrained	46.9	45	48	48	7.4	12.9	12.9
50-best ∪ CSLUt	75.0	84	**96**	**96**	11.9	21.1	21.1
50-best ∪ Reranker	72.1	87	90	89	12.1	20.9	20.8

Table 7.5: Average path-distance and path-length from the parser-best candidate to the oracle-best candidate; distance is measured as Levenshtein distance between sets of parse nodes. Paths follow the candidate-ordering as defined by the baseline parser-score (Base), pairwise-F score (Pair-F), or pairwise-D score (Pair-D).

lengths differ so much. We can surmise that the oracle candidate remains at about the same distance from the top-ranked candidate (according to either of the three ranking metrics). The path from the top-ranked candidate to the oracle-candidate is comprised of a larger number of steps under the similarity- or distance-ordering, but each of the steps themselves are shorter than the steps along the path created by following the parser-score ordering. Thus the similarity- and distance-ranking creates a denser space between the first candidate and the oracle-candidate.

There are many possible ways to order candidates in a list, and if the search space is truly a fully-connected graph, then any downstream stage would be able to move from one candidate to another without being restricted to some pre-defined path through the candidates. In Chapter 8 we will examine several ways in which candidate-order can be used to affect downstream stages, but for now we simply want a sense of how candidates are laid out in the space.

7.2.4 Nearest Neighbor Analysis

Having formally defined the area and density of a search space, we can now apply a *nearest neighbor analysis* [54, 188] on our seven data sets under investigation. Calculating the k-nearest neighbors of a data point is a common method in pattern recognition, typically used for clustering data points as discussed in Section 3.2.9. We will use the nearest-neighbor analysis to measure the regularity of a search space.

The nearest neighbor statistic R is defined as follows:

$$R = R_o / R_e \tag{7.3}$$

where R_o is the observed average of the distance from each point in the space to its nearest neighbor, and R_e is the expected average distance if the same number of points were randomly scattered in the same area. We can define R_e as:

$$R_e = \frac{1}{2\sqrt{\rho}} = \frac{1}{2\sqrt{n/A}} \tag{7.4}$$

where ρ is density, n is the number of points in the space, and A is area. If the nearest neighbor statistic R=1, then we know that the data is scattered in a random pattern throughout the space. If R is much lower than 1 (i.e., R_o is much smaller than R_e), then the data points are more clustered than randomly distributed; if R is much larger than 1 (i.e, R_o is much higher than R_e), then the data points are more uniformly distributed than randomly.

Since we are interested in whether or not the candidates are distributed uniformly throughout the space, we will test a hypothesis of $H_0 : R \leq 1$ ($H_A : R > 1$). If H_0 is true, then the mean of R

Condition	n	**Density** (n/**Area**)	R (R_o/R_e)	
50-best	49.9	387.2	9.9	*
100-best	99.5	577.6	14.0	*
Self-trained	49.8	357.6	9.9	*
CSLUt-constrained	49.0	435.8	9.8	*
Reranker-constrained	48.8	442.6	9.8	*
50-best ∪ CSLUt	74.9	450.7	12.1	*
50-best ∪ Reranker	72.1	472.2	11.9	*

Table 7.6: Using the nearest-neighbor statistic to determine whether the parse candidates, as produced under various conditions, are distributed uniformly throughout the space; distance is measured as the D-score distance between sets of parse nodes.

is 1. Furthermore, according to Petrere [170], the variance of R is approximately:

$$\mathrm{Var}(R) = (4 - \pi)/(\pi * n) \tag{7.5}$$

where n is the number of points. So we can standardize R:

$$z = (R - 1)/\sqrt{\mathrm{Var}(R)} \approx 1.913 * (R - 1) * \sqrt{n}. \tag{7.6}$$

The distribution of z should be approximately normal, so values of z larger than 1.68 will allow us to reject H_0 in favor of H_A at p<0.05 [179]; in other words, we can conclude that the candidates are *not* randomly distributed in the space.

In Table 7.6 we apply this nearest neighbor statistic to our seven data sets. The nearest neighbor of each parse tree was calculated using our D-score distance metric from Section 7.2.1.[8] The density of each candidate set has been re-calculated (from what was reported previously in Table 7.3) in terms of the D-score distance, but the relative comparisons remain: the 100-best set has the highest density, and the 50-best and Self-trained sets have the lowest. As we can see from the table, the nearest-neighbor statistic is significantly greater than one for all of our data sets, indicating that, somewhat surprisingly, the data points are not randomly distributed throughout the space and, in fact, are uniformly distributed.

7.2.5 Spatial Characteristics of Other Constraint Types

The examples that we have used throughout this chapter to measure the spatial characteristics of a candidate set have all been examples of fully-specified, complete solution constraints. Thus our measurements gave an exact characterization of the space to be searched by our downstream reranker. However, one could also apply these metrics to under-specified or partial solution constraints to obtain an approximate characterization of the downstream stage's search space.

7.3 Manipulating the Distribution of Candidates

In the remainder of this chapter we will explore the effects of manipulating the distribution of candidates in a space. We have discussed the pros and cons of the "diversity" of a space; in Section 7.3.1 we will experiment with one method to maximize diversity using the distance metrics

[8] With our approximation to a two-dimensional space here, the R statistic performs better with continuous numbers, such as those produced by the D-score, than with a small set of discrete numbers, such as those produced by the Levenshtein edit distance. We did, however, repeat the test with the Levenshtein distance calculations, which returned nearly identical results.

we discussed in Section 7.2.1. We hypothesized that a large distance between candidates might prove problematic for downstream learning algorithms, so in Section 7.3.2 we will minimize step-distance in an n-best list of parses by applying the Levenshtein distance metric to parse trees as discussed in Section 7.2.2. Finally, in Section 7.3.3 we explore a method to generate candidates in a systematic stepwise manner, to ensure complete coverage of the space.

We used the Charniak and Johnson [44] parser and reranker for all of the experiments conducted in this section, with exactly the feature sets described in that paper. The reranking model was trained on output from the Charniak parser on WSJ sections 2-21, with section 00 used as heldout. The output from the Charniak parser was produced using a typical 20-fold cross-validation scheme with 2000 sentences per fold. We report results on section 24. Each of the experiments differ only in the data that was input to the reranker for training and testing, essentially inserting another processing stage S_x between the parser and reranker stages in the pipeline. The different processes implemented at this stage S_x will be discussed in each section below.

7.3.1 Maximizing Diversity

The objective of this section is to determine how altering the diversity of a search space will affect downstream performance. We will, of course, report on reranker-best F-score, as well as oracle-best and oracle-worst rates. However, we will *also* examine the spatial effects of the manipulation, by applying the graph-based metrics introduced in Section 7.2.3 to the manipulated space.

Here we take a simple approach to maximizing diversity in a space, by selecting from a large candidate set only those that meet our diversity criteria. Specifically, we start with the 100-best parses output by that Charniak parser and select out of those 100 only 50 parses, so we can compare to the baseline of 50-best parses output directly by the parser. We conduct a full pairwise comparison of the candidates in the 100-best list, using the F-score similarity metric discussed in Section 7.2.1 to measure the similarity between two parse trees, then rank the 100 candidates according to the averaged pairwise score. The top candidate—where "top" is defined either (1) as the parser-best candidate, (2) as the maximally-similar node, at the center of the space, or (3) as the maximally-diverse (or minimally-similar) node, at the periphery of the space—is used to initialize the newly defined, manipulated space. Then candidates are selected from the 100-best parse list according to their pairwise similarity or diversity score, and the process continues until the new space is full (at 50 candidates) or all of the 100-best candidates have been selected.[9]

Table 7.7 shows the results of this experiment. None of the new techniques improved over our baseline. However, there are several items of note in this table. First, the oracle-worst rate is 10 percentage points higher for the maximized-similarity sets than for the baseline; whereas the oracle-worst rate for the maximized-diversity sets are about 5 points lower than the baseline. Second, none of the oracle-best rates were affected much by the manipulation. Decreased oracle rates are often "blamed" for poor downstream performance, as we discussed in Chapter 6, but the oracle-best rates did not drop much here and yet reranker performance did not improve. Finally, note the increase of 7.7 percentage points from parser-best to reranker best under the maximized-diversity condition where the initialization point was at the edge of the space, then compare to the much smaller gain of 1.3 points achieved under the baseline condition.

[9] The Charniak parser occasionally outputs fewer than n candidates for an n-best list.

[10] Note that because we select candidate from the 100-best parse candidates, the parser-best score is not identical to the parser-best score under the baseline 50-best condition, but instead matches the parser-best score under the 100-best condition, reported elsewhere.

Conditions	**Parser Best**	**Initial Best**	**Reranker Best**	**Oracle Best**	**Oracle Worst**
Baseline (50-best)	88.9	–	90.2	96.0	68.1
Maximized Similarity					
Init parser-best[10]	88.8	88.8	89.7	94.0	78.1
Init centroid (max-sim)	89.1	89.3	89.9	94.0	78.4
Maximized Diversity					
Init parser-best[10]	88.8	88.8	89.7	94.9	64.6
Init centroid (max-sim)	87.3	89.3	89.2	95.5	64.6
Init edge (max-div)	80.4	69.6	88.1	95.1	64.6

Table 7.7: F-scores of the parser-best, reranker-best, and oracle-best candidates from the top-50 parse candidates selected from the 100 parser-best candidates; the top-50 were selected based on minimal or maximal distance from either the parser-best candidate, the centroid of the 100-best space (the maximally-similar candidate), or the edge of the 100-best space (the maximally-diverse candidate).

Here we briefly discuss why we only maximize *diversity* when initializing the space with an edge node, since maximizing similarity from such a starting point would result in a set of candidates clustered right at the edge of the space, as far away as possible from the center of the space. The centroid is generally a very strong candidate, as seen by the Initial-Best column, which provides a score of the "top" candidate used to initialize the manipulated space—in fact, the centroid candidate has a slightly higher F-score than the parser-best candidate. Thus, generating a cluster of candidates far away from such a strong baseline candidate simply seemed illogical.

Having conducted the typical evaluations for our diversified candidate sets, we now examine the effects on the spatial characteristics of the space. In Table 7.8, we can see the diameter, radius, density, and path-distance characteristics of our newly-defined sets. As expected, the diameter and radius[11] of the maximized-similarity sets decreased as compared to the baseline, and correspondingly the density of the set increased. The path-distance characteristics, too, are as expected: shorter paths in the highly-similar sets, and longer paths in the highly-diverse sets. One slightly odd result can be seen with the edge-initial node, which is nearly as dense as the highly-similar sets.

Conditions	**Avg Diam.**	**Avg Radius**	**Density (n/Area)**	**Path-Distance**	
				to n	**to Oracle**
Baseline (50-best)	17.7	11.0	0.19	355	63
Maximized Similarity					
Init parser-best	10.4	6.4	0.56	248	40
Init centroid (max-sim)	10.2	6.3	0.57	243	39
Maximized Diversity					
Init parser-best	20.3	13.7	0.12	494	105
Init centroid (max-sim)	20.3	13.3	0.13	494	108
Init edge (max-div)	20.3	9.3	0.47	495	156

Table 7.8: Diameter and radius, diversity, and path distances, measured as Levenshtein distance between sets of parse nodes, of the top-50 parse candidates selected from the 100 parser-best candidates; the top-50 were selected based on minimal or maximal distance from either the parser-best candidate, the centroid of the 100-best space (the maximally-similar candidate), or the edge of the 100-best space (the maximally-diverse candidate).

[11] Recall that the diameter and radius are both based on eccentricity, which is the maximum distance from one node to any other in the graph; the diameter is the maximum eccentricity of the graph and the radius is the minimum eccentricity.

7.3.2 Minimizing Step-Distance

In this section we will examine how "step-size," or the distance between two points in a space, can affect downstream performance. As before, we will report on reranker-best F-score and oracle rates, as well as the spatial characteristics of the manipulated space.

For this set of experiments we began with the 50-best candidates output by the Charniak parser. Initializing always with the parser-best candidate, we then included any point in the 50-best set that was within some bounded distance of a point in our newly defined, manipulated space. We used the Levenshtein distance metric to calculate the distance between any pair of parse trees—note that we use the raw Levenshtein distance here, not an averaged pairwise comparison score. Table 7.9 shows the results of these experiments; we do see a small but statistically insignificant gain in reranker-best accuracy at the Levenshtein-distance threshold of 9.

Conditions	n	**Parser Best**	**Reranker Best**	**Oracle Best**	**Oracle Worst**
Baseline (50-best)	49.9	88.9	90.2	96.0	68.1
Lev-distance 1	23.4	–	89.2	90.8	87.9
Lev-distance 2	33.1	–	89.4	92.0	86.0
Lev-distance 3	40.5	–	89.6	93.4	83.5
Lev-distance 4	44.1	–	89.8	94.1	81.4
Lev-distance 5	45.6	–	90.0	94.5	79.4
Lev-distance 6	47.6	–	90.2	95.0	76.9
Lev-distance 7	48.2	–	90.3	95.2	75.4
Lev-distance 8	48.8	–	90.3	95.3	73.4
Lev-distance 9	49.2	–	**90.4**	95.5	72.5
Lev-distance 10	49.5	–	90.3	95.7	71.3
Self-trained	49.8	90.2	90.9	96.4	67.0
Lev-distance 1	21.9	–	90.3	91.8	83.7
Lev-distance 2	31.7	–	90.4	93.0	79.9
Lev-distance 3	40.1	–	90.6	94.4	76.0
Lev-distance 4	43.8	–	90.7	95.0	73.9
Lev-distance 5	45.6	–	90.7	95.3	72.4
Lev-distance 6	47.4	–	90.5	95.7	70.7
Lev-distance 7	48.0	–	90.7	95.8	70.1
Lev-distance 8	48.7	–	90.8	96.0	69.2
Lev-distance 9	49.0	–	90.7	96.1	68.8
Lev-distance 10	49.3	–	90.8	96.3	68.3
50-best ∪ CSLUt	75.0	89.4	90.3	96.6	65.6
Lev-distance 1	32.7	–	89.6	91.5	82.0
Lev-distance 2	47.0	–	89.7	92.8	78.2
Lev-distance 3	58.5	–	89.8	94.2	74.7
Lev-distance 4	64.1	–	89.9	94.8	72.5
Lev-distance 5	67.0	–	90.0	95.3	71.1
Lev-distance 6	70.2	–	90.3	95.8	69.3
Lev-distance 7	71.4	–	90.3	96.0	68.6
Lev-distance 8	72.5	–	90.2	96.2	67.8
Lev-distance 9	73.2	–	90.2	96.3	67.3
Lev-distance 10	73.9	–	90.3	96.5	66.8

Table 7.9: F-scores of the parser-best, reranker-best, and oracle-best thresholded parse candidates selected from the 50-best parser candidates. Parse trees are included in the thresholded set if the parse tree's Levenshtein distance from any other tree in the thresholded set is less than or equal to the threshold.

Conditions	**Avg Diam.**	**Avg Radius**	**Density (n/Area)**	**Path Distance** to n	**Path Distance** to Oracle
Baseline (50-best)	17.7	11.0	0.19	355	63
Threshold					
Lev-distance 1	7.5	3.5	0.97	91	8
Lev-distance 2	10.0	4.5	0.92	156	19
Lev-distance 3	12.2	5.3	0.86	221	31
Lev-distance 4	13.7	5.8	0.86	261	40
Lev-distance 5	14.5	6.1	0.86	282	44
Lev-distance 6	15.6	6.4	0.85	310	49
Lev-distance 7	16.0	6.6	0.85	320	52
Lev-distance 8	16.5	6.7	0.85	331	55
Lev-distance 9	16.8	6.8	0.84	338	57
Lev-distance 10	17.1	6.9	0.85	344	60

Table 7.10: Diameter and radius, diversity, and path distances, measured as Levenshtein distance between sets of parse nodes, of the thresholded parse candidates selected from the 50-best parser candidates. Parse trees are included in the thresholded set if the parse tree's Levenshtein distance from any other tree in the thresholded set is less than or equal to the threshold.

Another note of interest is the progression of oracle-best and oracle-worst rates. As the threshold increases, allowing more of the 50-best candidates to be included in the space, the oracle-best rate increases while the oracle-worst rate decreases. The rate of increase and decrease is not identical, but is certainly a much smoother progression than we have seen for other methods of pruning a space, including simply taking the top-n parses as ranked by the parser probability score.

In Table 7.10 we see the spatial characteristics of our manipulated sets. The most noteworthy result from this table is that of the density, which is very high at all thresholds, even at a threshold of 10 where nearly all of the 50-best candidates have been included. Furthermore, we can see that while the diameter of the space at threshold-10 is nearly identical to that of the baseline condition, the radius is much smaller. Thus we make the supposition that the Levenshtein-distance threshold is perhaps excluding some of the outliers, very distant candidates that can greatly affect measurements of the space.

7.3.3 Generating a Complete-Coverage Search Space

The experiments presented in both of the previous sections suffer from the same potential problem: we selected candidates from a set generated by a parser, which is trained to produce a high-quality top candidate, not a high-quality *set* of candidates. In fact, a typical reranking system might suffer from a similar problem. Furthermore, the output candidate set may have qualities that prove difficult for reranker training and testing.

Selecting the method for and then defining the search space in a pipeline is typically ad-hoc, often based on the implementations of other pipeline systems within the same field. For example, many recent pipeline systems have been implemented with 50-best or 20-best lists to define the search space of a pipeline stage, with very little reasoning as to why 50 (or 20) was selected as the size of the list, other than citations to previous studies implementing the same-size lists in similar pipeline systems. Current practices for choosing the size of an enumerated set of solutions include holding the size of the solution set constant across each problem in a data set. Intuitively, however, it might be reasonable to output a larger search space for problems with greater ambiguity, and a smaller set to represent the candidates for a simpler problem, such as that presented by annotating or translating a shorter word sequence. This research will argue for and present better methods for defining a (fully-specified) solution set beyond selecting some arbitrary value of n for an n-best list.

Tree Transformation

In this section we will explore a different method for generating a list of candidates for reranking. Namely, we will use a transformation (or transduction) technique to define a complete-coverage search space (discussed in Section 7.1.4). Other research has examined methods like using minimum Bayes' risk [87] for training the upstream system. Our algorithm generates n-best lists like confusion matrices, based on the differences between an initial parse candidate and some target parse coverage.

The tree-transformation technique begins by representing parse trees as a set of labeled brackets, as we did in Section 7.2.2 to calculate Levenshtein distances between parse trees. Once we have represented a tree as a sequence of brackets,[12] we can simply apply finite-state string transducers to the sequence. By using this finite-state sequence representation, as opposed to a context-free tree representation, we can take advantage of well-known sequence transduction algorithms. Thus we define three classes of allowable edits: insertions, deletions, and substitutions. Insertions add a labeled bracket to the existing set, deletions remove a bracket from the set, and substitutions replace one of the labeled brackets in the set with a new one. We modified the tree-alignment software `ParseDyff`,[13] to perform a minimum-edit distance comparison from the parser-best output from the Charniak [41] parser to the centroid parse in WSJ sections 2-21, section 00, and section 24.[14] We extract from `ParseDyff` the set of edits necessary to transform one source tree into its target tree.

Table 7.11 shows the results of two experiments. For the first set of experiments, we defined our n-best lists of parses to consist of just the parser-best parse and the centroid parse. These results are shown in the table as "1,centroid." Next, in order to explore the effects of a *fully-covered space*, we applied our tree-transformation algorithm described above to generate all parses between the parser-best parse and the centroid pase. Results from these experiments are shown in the table as "1-to-centroid" and referred to as "full-coverage." From the second column of the table (n), we see that the centroid and the one-best parse are often identical, since the average size of the 1,centroid lists is less than two. Furthermore, since the 1-to-centroid lists are also less than two on average (except for the 50-best ∪ CSLUt-constrained condition), we now know that the centroid and the one-best parse are not very "distant" in general. In fact, the average distance from the one-best to the centroid tells us something about the space defined by the candidate set; of particular interest is the 50-best ∪ CSLUt-constrained condition, where the size of the 1-to-centroid lists tells us that the distance between the center of the space and the parser-best candidate is farther than in other conditions. This makes some intuitive sense, since this condition is the union of two arguably very different lists, where the constraints used to generate the second list were derived from a model with very different features and with a different optimization objective. Thus our tree-transformation algorithm serves not only to cover any gaps in the space, but also produces a measurement that can be used to characterize the space in a novel way.

We can also see from the table that the 1-to-centroid lists are larger than the 1,centroid lists, indicating that there were gaps in the space between the two parses; we used our tree-transformation algorithm to fill in those gaps and generate a fully-covered space. The full-coverage condition resulted in a statistically significant reranker improvement, as compared to the the 1,centroid condition, under three conditions (marked with an * in the table). Interestingly, under two of those

[12] By choosing to represent our parse trees as a non-hierarchical sequence of brackets, and performing edit operations on these sequences, we might encounter difficulties with crossing brackets and deleted preterminals or words. To address this problem (which did arise but only occasionally), a simple post-processing step was added to check for such issues, and remove invalid sequences from the output set.

[13] http://www.cs.brown.edu/∼dmcc/software/parsedyff/

[14] The output from the Charniak parser was, as mentioned previously, produced using a typical 20-fold cross-validation scheme.

Conditions	n	**Parser Best**	**Reranker Best**	**Oracle Best**	**Oracle Worst**	
50-best	47.9	88.9	90.2	96.0	68.1	
1,centroid	1.42	–	89.6	90.6	87.8	
1-to-centroid (full-coverage)	1.75	–	89.6	90.6	87.8	
1-to-oracle	1.62	–	93.7	96.1	88.9	
1-to-reference	1.80	–	96.9	100.0	88.9	
100-best	95.2	88.8	90.1	96.4	64.6	
1,centroid	1.46	–	89.5	90.5	87.5	
1-to-centroid	1.87	–	89.5	90.7	87.5	
Self-trained	49.8	90.2	90.9	96.4	67.0	
1,centroid	1.39	–	90.5	91.6	89.1	
1-to-centroid	1.72	–	**90.7**	91.7	89.0	† *
CSLUt-constrained	46.2	88.4	89.5	94.1	69.1	
1,centroid	1.41	–	88.9	89.8	87.2	
1-to-centroid	1.69	–	89.0	89.9	87.2	
Reranker-constrained	46.9	89.6	90.5	95.1	69.9	
1,centroid	1.40	–	90.0	90.9	88.5	
1-to-centroid	1.63	–	90.1	90.9	88.5	
50-best ∪ CSLUt	75.0	89.4	90.3	96.6	65.6	
1,centroid	1.54	–	89.9	91.4	87.5	
1-to-centroid	**2.26**	–	**90.2**	91.7	87.4	† *
50-best ∪ Reranker	72.1	89.2	90.6	96.5	66.4	
1,centroid	1.46	–	89.9	91.1	88.0	
1-to-centroid	1.92	–	**90.1**	91.3	87.9	*

Table 7.11: F-scores on WSJ section 24 where the search space is defined as the parser one-best and centroid parses (1,centroid), or the fully-covered space generated by step-wise edits from the one-best to the centroid parse candidate (1-to-centroid). * indicates a statistically significant improvement of the (1-to-centroid)-trained reranker over the (1,centroid)-trained reranker (paired t-test, p<0.05). † indicates a statistically *insignificant* difference between the baseline n-best–trained reranker and the reduced-space reranker (paired t-test, p<0.05).

three conditions (marked with a † in the table), the full-coverage 1-to-centroid reranker output resulted in a statistically *insignificant* difference from the full n-best reranker output.

Under *all* conditions, the reranker provides a statistically significant improvement (paired t-test, p<0.001) over the matching parser-best output; a noteworthy result, considering that the experimental conditions resulted in n-best lists consisting of fewer than 2 candidates on average, in all except one condition. Note that training reranker models with such small candidate sets requires more than an order of magnitude less time than training models with the full candidate sets. Thus, generating a full-coverage space proved beneficial for downstream reranker performance, even when starting from such a small space as that defined by the one-best and centroid candidates.

Now let us briefly discuss two "cheating" experiments, shown in Table 7.11 as the third and fourth row under "50-best," in which we used (a) the reference parse as the target tree in the tree transformation, and (b) the oracle parse as the target tree. These results are shown as "1-to-reference" and "1-to-oracle," respectively, and are presented here because they show some promise on several fronts. Note that the Charniak and Johnson [44] reranker was able to achieve a much higher F-score with the very small lists generated by tree transformation. Part of this success is likely due to the fact that under the cheating conditions, we removed much of the noise in the n-best lists: if the parser's top candidate is also the oracle candidate, then the size of that list produced under the tree transformation "to-oracle" condition will be one, and the reranker must

Conditions	n	**Parser Best**	**Reranker Best**	**Oracle Best**	**Oracle Worst**	
50-best	47.9	88.9	90.2	96.0	68.1	
1,centroid2	2.00	–	89.5	90.8	87.1	
1-to-centroid2 (full-coverage)	2.61	–	89.6	90.9	86.9	
100-best	95.2	88.8	90.1	96.4	64.6	
1,centroid2	2.00	–	89.5	90.8	86.8	
1-to-centroid2	2.77	–	89.4	90.9	86.6	
Self-trained	49.8	90.2	90.9	96.4	67.0	
1,centroid2	2.00	–	90.6	91.9	88.2	
1-to-centroid2	2.64	–	**90.7**	92.0	88.0	† *
CSLUt-constrained	46.2	88.4	89.5	94.1	69.1	
1,centroid2	2.00	–	89.0	90.0	86.6	
1-to-centroid2	2.63	–	88.9	90.2	86.4	
Reranker-constrained	46.9	89.6	90.5	95.1	69.9	
1,centroid2	2.00	–	90.0	91.2	87.6	
1-to-centroid2	2.47	–	90.0	91.3	87.6	
50-best ∪ CSLUt	75.0	89.4	90.3	96.6	65.6	
1,centroid2	2.00	–	90.1	91.7	86.6	†
1-to-centroid2	**3.16**	–	90.1	92.0	86.4	†
50-best ∪ Reranker	72.1	89.2	90.6	96.5	66.4	
1,centroid2	2.00	–	90.0	91.4	87.1	
1-to-centroid2	2.70	–	90.1	91.6	87.0	

Table 7.12: F-scores on WSJ section 24 where the search space is defined as the parser one-best and second-centroid parses (1,centroid2), or as the fully-covered space generated by step-wise edits from the one-best to the second-centroid parse (1-to-centroid2). * indicates a statistically significant improvement of the (1-to-centroid2)-trained reranker over the (1,centroid2)-trained reranker (paired t-test, $p<0.05$). † indicates a statistically *insignificant* difference between the baseline n-best–trained reranker and the reduced-space rerankers (1,centroid2 or 1-to-centroid2) (paired t-test, $p<0.05$).

therefore select that candidate.

Table 7.12 is nearly identical to Table 7.11, with the exception of how the centroid was calculated. As we noted earlier, in many of the n-best lists, the one-best parse was *also* the centroid, and thus many of the lists consisted of just one candidate. The benefit of a reranker is lost in such scenarios, since there is nothing to rerank. Therefore, in Table 7.12, we changed the method by which we calculated the centroid, in order to guarantee that there would be at least 2 candidates in the set input to the reranker. In these experiments, we excluded the one-best parse from the space, then calculated the centroid of the resulting space; we call this the second-centroid. The second column (n) of Table 7.12 shows that, indeed, each of our candidate sets contain at least two candidates. Furthermore, the increase in the number of candidates from the 1,centroid2 condition to the full-coverage 1-to-centroid2 condition is larger than the increase seen in Table 7.11, which is unsurprising; like the reranker, our tree-transformation algorithm does no work when given a single-candidate set, and necessarily produces more output when given a two-candidate set. Interestingly, the number of candidates under the full-coverage set under the 50-best ∪ CSLUt-constrained condition is again higher than the other conditions, confirming that the distance from the one-best parse to the center of the space is farther under this condition.

Under each experimental condition, the oracle-best rate is higher than in Table 7.11, a benefit of increasing the size of the initial search space, though the oracle-worst rates decrease under each condition as well. Note that in this table, there is no longer a statistically significant improvement from the 1,centroid2 space to the full-coverage 1-to-centroid2 space under the 50-best $\cup$ CSLUt-constrained and 50-best $\cup$ Reranker-constrained conditions; the Self-trained condition retains its statistically significant improvement (marked by an * in Table 7.12). Three of the experimental conditions are statistically insignificantly different from training with the full n-best lists (marked by a † in the table), as compared to the two conditions in Table 7.11: the Self-trained 1-to-centroid2 condition, the 50-best $\cup$ CSLUt-constrained 1-to-centroid2 condition, and additionally the 50-best $\cup$ CSLUt-constrained 1,centroid2 condition.

There are several points to be drawn from the experiments conducted in this section. First and foremost, we were able to achieve reranker improvement over parser output with very small candidate sets. Second, none of the transformation-generated lists include the parser's score for any of the candidates, which has been cited as one of the most important features in the reranker [44]; yet the reranker was still able to improve significantly over the parser-best output. Third, the one-best candidate and the candidate at the center of the defined search space were of fairly high quality; a question for future research is whether this level of quality would hold true for candidate sets produced by other systems. Finally, by generating additional parse candidates "between" two high-quality candidates to produce a full-coverage space, we were able to achieve an additional improvement in reranker performance.

7.4 Conclusion

While the methods for manipulating the distribution of candidates in a space did not yield significant improvements in one-best parsing accuracy, we did make two interesting discoveries. First, our simple method for manipulating the diversity of a candidate set clearly affected the density of the manipulated sets, indicating that density may be a reasonable metric to formally measure diversity. Second, the distance between candidates in the set inarguably affects oracle rates, both oracle-best and oracle-worst, as Table 7.9 clearly demonstrated. On the other hand, the relation between oracle-rates and reranker performance remains unclear.

The goal of this chapter was to capture the spatial characteristics of a set of candidates. With the metrics defined in Section 7.2, particularly the novel application of graph-theory metrics to characterize the distribution of candidates in a space, we feel that this goal has been achieved. By defining formal metrics to measure the area and density of a space, this chapter has opened the door for quantifiable comparisons across different data sets, and indeed, across different application areas.

Chapter 8

Weighted Constraints

Chapters 6 and 7 discussed fairly concrete characteristics of pipeline constraints. This chapter will concentrate on a different dimension of the constraint space: a probability distribution[1] over the constraints, typically output as model scores in the upstream stage. If we envision the search space as a two-dimensional area, then a probability distribution over the constraints would add a third dimension.

A weighted distribution over the search space can significantly impact downstream performance, as we have mentioned numerous times throughout this dissertation. Many systems include a model score with each output; this model score may then be used by downstream stages as a weighted distribution over their search space. For example, in parsing, many systems include the parser probability score as a feature in the next-stage reranker, including the Collins [64] parser and the Charniak and Johnson [44] parsing pipeline. This probability score is often reported to be a highly-informative feature [44, 68], and thus is used frequently in pipeline systems. As an overall characteristic of the search space, the distribution is often disregarded (beyond identifying the highest-scoring one-best candidate) but can, as will be shown in this chapter, prove problematic.

In Section 8.2 we will discuss a number of different ways to characterize a probability distribution, including how peaked it is, and how closely it approximates a normal distribution. In characterizing a weight distribution we might also ask how many peaks the distribution has, the shape of the distribution (as determined by its mean and standard deviation), and whether or not the distribution is symmetrical. In Section 8.3 we discuss two functions for comparing two probability distributions, and find that the relative comparison of two distributions can be more informative than characterizing each distribution separately. Finally, in Section 8.4 we examine the ways in which probability distributions are utilized in pipeline systems, and experiment to determine how downstream pipeline performance is affected by a weighted distribution over the input constrained space. Results show that the peakedness of the probability distribution accounts for some of the puzzling results from earlier in the dissertation; furthermore, the effects of a peaked distribution can be mitigated using a simple empirical optimization technique.

8.1 Alternate Distributions

Throughout this chapter we will be using the Pairwise-F similarity metric and the Pairwise-D distance metric introduced in Chapter 7 (p. 102) to re-score and re-order lists of parse candidates.

[1] The weights assigned to constraints in a space are not necessarily probabilities; we refer to them as such for convenience sake, and to avoid confusion with the learned weights of features in a model. Section 8.2.1 will discuss methods for transforming any weight distribution into a probability distribution.

We will compare the characteristics of the distributions defined by these scoring metrics to the characteristics of the probability distribution output by the Charniak and Johnson [44] parser.

Our motivation for using these metrics in comparison to the parser probability score is twofold. First, the relative ranking of candidates within the list as defined by the parser-probability distribution may not be an accurate reflection of the relative similarities between the ranked candidates. Recall that the parsing model is optimized to find the maximum likelihood candidate; similarly the reranking model optimizes with respect to the conditional likelihood of the data. There is no reward (or penalty) for determining an accurate ranking of the full list of candidates. Thus by using characteristics of the list itself, of the similarities and distances between candidates in the list, we may be able to create a weighted distribution that more accurately reflects the qualities of the underlying space.

Second, it may be that the parser probability scores are not the most reliable metric derivable from the baseline system in terms of assessing the quality of a candidate. Thus we will examine using the Pairwise-F and Pairwise-D scores as weight-distribution features that go beyond the parser probability score to capture information about similarity, rank, or centrality of the particular candidate within the set output by the baseline parser.

8.2 Peakedness Metrics

One way in which we may characterize a probability distribution is according to how *peaked* or *flat* it is. The flattest distribution is, of course, one which places equal probability on each possible solution or constraint in the space. On the other extreme, a very peaked distribution would be one that places most of the probability mass on a single constraint in the set, or a single solution in the space. Either extreme might be detrimental to pipeline performance: if the probability distribution is very peaked, then downstream stages may be essentially restricted to only those constraints near the distribution peak; on the other hand, if the distribution is nearly uniform, then its utility as a feature is degraded since it provide less discriminative information about the constraints for downstream stages.

Previous work in parsing [152] has assumed that a more peaked distribution indicates greater confidence from the upstream stage, but this "confidence" may not be well-founded. In fact, Gabbard et al. [91] claimed that the work done by Blaheta [25] on joint modeling context-free parsing and function tagging was unsuccessful due to the *peakedness* of the parser's probability distribution. Unfortunately most references to peaked distributions in the NLP literature do not provide quantitative measurements of the distributions' peakedness, relying instead on ad-hoc measurements of how much of the probability mass is placed on the top-ranked output candidate.

In this section we will first discuss this ad-hoc method for estimating distribution peakedness, then describe a method to more formally measure peakedness. For each of the distribution characterizations performed in this section, we calculate the characteristic on the distribution over each n-best list produced for WSJ section 24, then average the values across the entire set, for each of our seven reported conditions.

8.2.1 Normalization

One common estimation of distribution peakedness is to calculate how much of the normalized distribution is placed on the top-ranked candidate in the set. *Normalizing* the distribution consists of constraining the sum of all the probabilities in the distribution to be 1. For the parser-probability distribution, output as negative log probabilities, we divide the exponent of each output score by the sum of the exponents of the scores in the n-best list. For the pairwise scores, which range from [0..1], we divide each score by the sum of scores in the n-best list. Note that with the Pairwise-F

Condition	Parser-Score	Pairwise-F	Pairwise-D
50-best	0.79	0.15	0.16
100-best	0.80	0.11	0.11
Self-trained	**0.90**	0.15	0.16
CSLUt-constrained	0.82	0.15	0.16
Reranker-constrained	0.83	0.15	0.16
50-best ∪ CSLUt	0.80	0.13	0.13
50-best ∪ Reranker	0.80	0.13	0.13

Table 8.1: Average normalized probability of the top-ranked candidate of various probability distributions.

score, parses with values closer to 1 are ranked higher because they are considered to be more similar to the rest of the list. On the other hand, the Pairwise-D score ranks candidates with values nearer to 0 higher because they are considered to be closer to the rest of the list; thus we use 1−D in the calculations below.

Table 8.1 presents the average normalized probability of the top-ranked candidates for each of our conditions under investigation, where the candidates are ranked according to parser-probability, Pairwise-F, or Pairwise-D scores. We can see from the table that, as expected, the parser probability score does place a large portion of the distribution on the top-ranked candidate. Of particular note is the Self-trained condition, with nearly 10% more of the normalized distribution on the top-ranked candidate than any of the other conditions; a similar result was reported by McClosky et al. [153]. The pairwise distributions place much less of the normalized probability on the top-ranked candidate as compared to the parser-probability distribution. In a closer examination of the pairwise scores, we saw a high frequency of two-, three-, and even four-way ties for the top rank.

While this estimate certainly provides some interesting information, we do not feel that it is a rigorous enough metric. For example, while we now know that 79% of the parser probability mass is placed on the top candidate in the 50-best space, we do not have any information about how the probability has been spread across the remainder of the space. If the second-best candidate were receiving 20% of the probability mass, that would be a very different distribution than if the second-best candidate received an equal portion of the probability mass as the remainder of the candidates. The next section presents a peakedness metric which better captures this overall characterization of the distribution.

8.2.2 Kurtosis

Excess kurtosis, a metric drawn from probability theory and statistics, is defined as a measure of the peakedness of a distribution. Formally, kurtosis is a normalized form of the fourth central moment of a distribution, or the fourth moment around the mean divided by the square of the variance of the probability distribution:

$$k = \frac{\mu^4}{\sigma^4} - 3 \tag{8.1}$$

where μ is the mean of the distribution and σ is the standard deviation. Excess kurtosis has a lower limit of -2; there is no upper limit on excess kurtosis and it may be infinite. A normal distribution has an excess kurtosis of 0; a uniform distribution has an excess kurtosis of -1.2. A high-kurtosis distribution has a higher, sharper peak than a normal distribution, while a low-kurtosis distribution has a lower, more rounded peak. A normal distribution is called *mesokurtic*, a highly-peaked distribution is called *leptokurtic*, and a flat-topped distribution is called *platykurtic*. Figure 8.1 provides a visual representation of these three types of distributions.

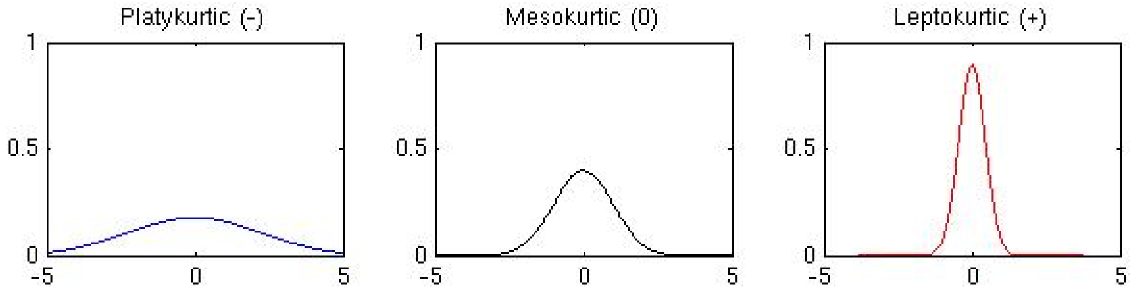


Figure 8.1: A visual representation of the three different types of kurtosis; the platykurtic distribution has a kurtosis of -1.5, the mesokurtic distribution has a kurtosis near 0, and the leptokurtic distribution has a kurtosis of 4.3.

Kurtosis can also be thought of as a measure of how outlier-prone a distribution is. A distribution that is more outlier-prone will have excess kurtosis greater than zero; a distribution that is less outlier-prone will have excess kurtosis less than zero.

In Table 8.2 we calculated the excess kurtosis[2] of the seven different parse-output conditions that we have been investigating, with the parser-probability distribution and the Pairwise-F and Pairwise-D distributions. One result of some interest is the kurtosis of the pairwise distributions under the Self-trained condition, which are lower (closer to normal) than any of the other values of kurtosis in the table. Overall, we can see from the table that the parser-probability distribution is very leptokurtic, i.e., extremely peaked. The pairwise distributions approach the kurtosis of a normal distribution. These conclusions confirm that the patterns we saw in Table 8.1 are consistent over the entire distribution.

Condition	**Parser-Score**	**Pairwise-F**	**Pairwise-D**
50-best	24.4	1.99	1.41
100-best	52.5	2.18	1.46
Self-trained	33.3	1.48	0.96
CSLUt-constrained	25.5	1.80	1.32
Reranker-constrained	26.3	1.86	1.40
50-best ∪ CSLUt	37.6	1.98	1.33
50-best ∪ Reranker	36.6	1.94	1.30

Table 8.2: Averaged kurtosis of various probability distributions.

8.3 Comparing Weight-Distributions

In the previous section we measured one characteristic of a distribution: its peakedness. There may be other characteristics of interest, but rather than compare distributions by one or two or three different characteristics, in this section we present two methods for directly comparing one distribution to another.

8.3.1 Kolmogorov-Smirnov Test

As a part of our kurtosis calculations we determined that the parser-probability distributions were quite a bit more peaked than the normal distribution. In this section we will directly compare each

[2] Calculated using the `kurtosis` function of MATLAB.[3] Since MATLAB does not subtract 3 as in Equation 8.1, we subtracted it from the result returned by the kurtosis calculation.

[3] http://www.mathworks.com/products/matlab/

distribution to the normal distribution. We expect that the distributions will not, in fact, be close to normal, but since the normal distribution is one of the simplest and most convenient to use, researchers are tempted to assume that a dataset is distributed normally, without rigorously justifying such an assumption. A normality test would provide that rigorous justification. Normality tests assess the likelihood that the given data set $\{x_1, \ldots, x_n\}$ comes from a normal distribution. Typically the null hypothesis H_0 is that the observations are distributed normally with unspecified mean μ and variance σ^2, versus the alternative H_a that the distribution is arbitrary.

One such normality test is the Kolmogorov-Smirnov (K-S) test, which may be used to determine whether two underlying one-dimensional probability distributions differ. The K-S test quantifies a distance between the empirical distribution function of the sample and the cumulative distribution function of the reference distribution. Here, our parser-probability and pairwise distributions will serve as the empirical distributions, which we will test against the normal distribution $\mathcal{N}(0,1)$. The null distribution of this statistic is calculated under the null hypothesis that the sample is drawn from the reference distribution.

Formally, the Kolmogorov-Smirnov statistic is:

$$D_{n,n'} = \sup_x |F_{1,n}(x) - F_{2,n'}(x)|, \tag{8.2}$$

where $\sup S$ is the least upper bound of set S, and $F_{1,n}$ is the empirical distribution functions of the empirical sample and $F_{2,n'}$ is the normal distribution function. The null hypothesis is rejected at level α if:

$$\sqrt{\frac{n \;\, n'}{n+n'}} D_{n,n'} > K_\alpha. \tag{8.3}$$

We ran the K-S test[4] over each of our seven test conditions, comparing each of the parser-probability and pairwise distributions to the normal distribution. All of the sentences in WSJ section 24 under the 50-best, 100-best, Self-training, 50-best $\cup$ CSLUt-constrained, and 50-best $\cup$ Reranker-constrained conditions differed significantly from the normal distribution. Interestingly, a few of the n-best lists produced under the constrained conditions (CSLUt-constrained and Reranker-constrained) matched the normal distribution: 22% of the parse lists under the pairwise distributions, and 37% of lists produced under the CSLUt-constrained condition and 67% of the Reranker-constrained lists.

8.3.2 Kullback-Leibler Divergence

In this section we explore the use of the Kullback-Leibler divergence statistic to compare two distributions. In probability theory, an f-divergence is a function $D_f(P\|Q)$ to measure the difference between two probability distributions P and Q. The divergence is intuitively an average, weighted by the function f, of the odds ratio given by P and Q. Thus for probability distributions P and Q of a discrete random variable, the Kullback-Leibler (KL) divergence of Q from P is defined to be:

$$D_{\mathrm{KL}}(P\|Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}. \tag{8.4}$$

One might be tempted to call the KL-divergence a *distance metric* for probability distributions, but this would not be entirely correct because the Kullback-Leibler divergence is neither symmetric—$D_{\mathrm{KL}}(P\|Q) \neq D_{\mathrm{KL}}(Q\|P)$—nor does it satisfy the triangle inequality. Still, the KL-divergence is a metric that will allow us to compare two probability distributions. Other measures of probability distance are the histogram intersection, Chi-square statistic, and the two-sample

[4] Using the `kstest` function of MATLAB.

Condition	$D_{\mathrm{KL}}(P\|F)$	$D_{\mathrm{KL}}(P\|D)$	$D_{\mathrm{KL}}(F\|D)$	$D_{\mathrm{KL}}(F\|P)$	$D_{\mathrm{KL}}(D\|P)$
50-best	1.8591	2.1902	0.0408	2.8024	3.2415
100-best	2.4055	2.7539	0.0370	3.7211	4.1899
Self-Trained	2.8363	3.2172	0.0443	**7.3208**	**8.1471**
CSLUt-constrained	2.0647	2.3879	0.0419	3.8610	4.3885
Reranker-constrained	2.1519	2.4805	0.0414	4.0839	4.6281
50-best ∪ CSLUt	2.2226	2.5724	0.0380	4.1015	4.7943
50-best ∪ Reranker	2.1739	2.5367	0.0414	3.6364	4.2691

Table 8.3: Kullback-Leibler divergence of different weighted distributions for datasets produced under various conditions.

Kolmogorov-Smirnov test.[5] We cannot use the Chi-square statistic to compare our distributions because this statistic assumes normal distributions, which as we showed previously, is nearly never true for the distributions we are interested in examining.

In Bayesian statistics the KL-divergence can be used as a measure of the information gain in moving from a prior distribution to a posterior distribution [177, 178]. Thus the KL-divergence is sometimes also called the *information gain* about X achieved if P can be used instead of Q. The KL-divergence may also be called the *relative entropy* for using Q instead of P.

In Table 8.3 we calculated the KL-divergence between each of our three distributions: parser-probability (P), Pairwise-F similarity (F), and Pairwise-D distance (D). In the middle column we see a confirmation of what we have empirically noted all along: the Pairwise-F and Pairwise-D distributions are nearly the same. In comparing the first two columns of results to the last two, we can see that the Pairwise-F and Pairwise-D distributions provide a greater information gain over the parser-probability distribution than the reverse cases. Of particular interest are the results in the last two columns under the Self-trained condition, noticeably higher than any of the other KL-divergence values. These results tell us that the Pairwise-F and Pairwise-D distributions provide the *greatest* information gain over the parser-probability distribution under the Self-training condition, which leads us to speculate that perhaps the parser-probability produced by the Self-trained model is, indeed, problematic.

8.3.3 Measuring Weighted Distributions in Different Pipeline Classes

If none of the downstream stages utilize the weight distribution as a feature, then it may seem that characterizing the distribution is a pointless exercise. However, the ability to characterize a distribution as a whole can also provide some information about the model itself, which is why distribution features are so useful in learning to search. In our work [162] (outside the scope of this dissertation), we built a mimic pipeline to estimate model-scores of an upstream rule-based stage which did not produce model weights [161]. Analyzing the weight-distribution output from such a mimic system might provide an interesting analysis of the underlying, mimicked model.

8.4 Learning with Weighted Constraints

Reranking is one of several commonly-used strategies in pipeline systems which makes use of the probability distribution output by an upstream stage as a feature to improve the reranker's search. Reranking with probability distribution features has proven to be a successful strategy for many NLP tasks such as parsing [64, 44], speech recognition [84, 97, 67, 183], and machine translation [199, 166]. However, the success of the reranking strategy is not guaranteed. In Chapter 5 (p. 84)

[5] We used the one-sample Kolmogorov-Smirnov test in the previous section to compare our empirical distributions to a reference distribution.

we discussed two separate research efforts in syntactic parsing where promising results did not achieve typical improvements when a reranking model was trained to be applied to the new baseline systems' output. The pipeline techniques of "self-training" [152] and "pipeline iteration" [110] both resulted in a similar lack of expected improvements with reranking.

We hypothesize that the underperformance of these two systems can be partially attributed to the use of the baseline system's output probability distribution as a feature in the reranker. Most reranking systems benefit from including the score from the baseline system within the reranking model. In fact, the difference in reranking performance with and without the parser baseline score as a feature was explicitly demonstrated in [44]: with the baseline score feature, reranking improved the parser F-score (on WSJ section 24) from 88.9 to 90.2, whereas without the score feature the reranker provided a smaller F-score improvement, to 89.5. We confirm these results in Section 8.4.1. The results from [44] established what has become a standard practice of reranking: to utilize the probability distribution from upstream stages as a feature in the model.

However, there are a number of reasons why using an upstream model's probability distribution directly as a feature in the current model may not be the best practice. First, since the probability score is often fairly accurate in indicating high-quality solutions, it tends to provide a particularly reliable feature. Training with such a strong feature can result in weight under-training for the other features in the set [35, 203]. One method for dealing with this is to remove the baseline-score feature from the reranking model during training, then *empirically optimizing* a mixing weight for combining the baseline score with the reranker score; our results from implementing such a method are shown in Section 8.4.2. Second, in order to improve upon the baseline output, a reranking model must select something other than the upstream model's best candidate solution at least once; otherwise the one-best output of the reranker would be unchanged from the baseline. Thus, even though the upstream probability distribution provides a fairly reliable estimate of high-quality candidate solutions, the reranking model must "down-weight" the feature in order to move away from the upstream model's top-ranked solution. Other methods for representing the score, such as *quantization*, might be better suited for learning to weight such a score feature, which we will explore in Section 8.4.3.

The results presented in the next several sections will demonstrate that the underperformance reported by McClosky et al. [152] and Hollingshead and Roark [110] can be at least partially attributed to the method for representing and optimizing on a probability distribution feature. The techniques presented below make better use of the probability distribution, and thus move closer to the expected levels of reranking performance.

Empirical trials were conducted by training Charniak and Johnson [44] reranker models on the n-best lists produced under each of our investigated conditions. Crossfold validation (20-fold with 2,000 sentences per fold) was used to train the reranking models. In all cases, WSJ section 00 was used as heldout data and section 24 as the development set. Unless stated otherwise, all reported results will be the F-score of WSJ section 24. Evaluation was performed using `evalb` under standard parameterizations. All statistical significance tests were conducted using the stratified shuffling test [215].

8.4.1 Unweighted Constraints

In this section we will test the oft-cited theory that a feature based on the weight-distribution output by an upstream stage is a highly informative feature [44, 68]. To perform this test, we simply remove the distribution feature from the reranker's feature set, to see how well the model can perform without such an informative feature. We call these reranking models "unweighted" models. Table 8.4 shows unequivocally that reranker performance decreases when this feature is removed. In fact, under the Self-trained condition the unweighted reranking model was unable

Condition	**Parser Best**	**Reranker Best**	**Unweighted RR Best**
50-best	88.9	90.2	89.5
100-best	88.8	90.1	89.5
Self-trained	90.2	90.9	89.7
CSLUt-constrained	88.4	89.5	88.8
Reranker-constrained	89.6	90.5	90.0
50-best ∪ CSLUt	89.4	90.3	89.6
50-best ∪ Reranker	89.2	90.6	89.6

Table 8.4: Comparison of rerankers trained under the de-facto standard of training with the parser-probability distribution features (Reranker-best) and without (Unweighted RR-best).

to achieve even the parser-best accuracy. However, under the 50-best condition the unweighted reranker achieves nearly half the gain that was achieved by the standard reranker model over the parser-best output.

8.4.2 Empirical Optimization

In this section we experiment with two different methods for optimizing a probability distribution as a feature in a reranking model. The default method, and one that is taken by Charniak and Johnson [44], is to essentially learn a *scaling factor*, where a single weight is learned for all values of the feature. As an alternative method, we propose and evaluate an *empirical optimization* approach. In an empirical optimization approach, the score from the upstream model's probability distribution is not included as a feature in the reranking model during training; we train an "unweighted" reranking model as in the previous section. This reranking model is then combined with the probability distribution score, scaled by weights empirically optimized on held aside data. These experiments closely parallel the weighted sequence-intersection experiments in Chapter 5 (p. 83).

This approach has multiple possible benefits. First, removing the probability distribution feature from reranker training may improve performance due to weight under-training when the distribution feature is left in the model. Second, any improvements could be due to optimizing the mixing with respect to parse accuracy achieved, rather than with respect to conditional likelihood, as the reranking training does.

Table 8.5 presents the results of empirically optimizing baseline-score feature weights in a learned reranking model. The Baseline column shows the F-score accuracy of the baseline-best parse candidate, and the Scaling Factor column shows the F-score accuracy of the reranker-best

Condition	**Parser Best**	**Scaling Factor**	**Empirical Optimization**	**"Mis-trained" Reranker**
50-best	88.9	90.2	90.2	–
100-best	88.8	90.1	**90.4**	90.2
Self-Trained	90.2	90.9	**91.0**	91.1
CSLUt-constrained	88.4	89.5	89.5	89.3
Reranker-constrained	89.6	90.5	**90.6**	90.5
50-best ∪ CSLUt	89.4	90.3	**90.7**	90.6
50-best ∪ Reranker	89.2	90.6	90.6	90.5

Table 8.5: F-scores on WSJ section 24 of reranker-best parses where the weight for the baseline-score feature was either learned using the de-facto standard method (Scaling Factor) or empirically optimized on held aside data (Empirical Optimization).

parse candidate using the de-facto standard method of learning a feature weight for the baseline score within the reranking model. The bolded entries in the table indicate the conditions in which empirically optimizing the weight for the distribution feature resulted in an improved reranker-best output. Note in particular the 50-best ∪ CSLUt condition, which under Empirical Optimization improves 0.4 percentage points above the standard scaling method (a statistically significant improvement at $p<0.05$), *as well as* improving over the "mis-trained" reranker model (introduced in Chapter 5, p. 84, and reproduced in the final column of this table). There is no change in the F-score for the 50-best output, though at least the empirical optimization does not harm accuracy of the learned reranking model. The F-score on the Self-training output improves slightly from 90.9 for the default method to 91.0 under empirical optimization. These results demonstrate that although a probability distribution can act as a beneficial feature, it can also overwhelm the evidence provided by other features, and thus including a distribution feature in training can prove detrimental to performance.

Application to Alternate Distributions

The Pairwise-F and Pairwise-D scores may exhibit similar behavior to a parser probability score in terms of down-weighting and under-training other features in the model, so we will also compare the efficacy of our two methods for learning with distribution features using these two metrics to define the distribution feature.

Table 8.6 shows the results of using Pairwise-F and Pairwise-D metrics as features in a reranking model, and compares these results to the effects of empirically weighting the distribution features. In examining these results, we focus on the differences between the results using the scaling factor method and using the empirical optimization method. As with the previous set of results, note in particular the performance of the 50-best ∪ CSLUt-constrained condition. With the Pairwise-D score, the empirical optimization method once again provides a significant improvement of 0.4 F-score over learning a linearly-scaled weight; similarly with the Pairwise-F score, we see an improvement of 0.6 F-score using empirical optimization instead of the traditional scaling factor method. Compare, also, the results for this condition to the same in Table 8.5, and note that the empirically-optimized Pairwise-D distribution provides the highest observed reranker-best F-score for the 50-best ∪ CSLUt condition at 90.8.

Clearly, we have seen that empirically optimizing the weight for distribution features can result in significant performance improvements. However, we should also note that improved performance is not guaranteed: under the 50-best condition and using the Pairwise-D score as the distribution feature, the scaling factor method outperforms the empirical optimization by 0.4 points.

	Pairwise F-score		**Pairwise D-score**	
Condition	**Scaling Factor**	**Emp Optim**	**Scaling Factor**	**Emp Optim**
50-best	90.1	**90.2**	90.5	90.2
Self-trained	90.5	**91.0**	90.9	90.9
50-best ∪ CSLUt	90.1	**90.7**	90.4	**90.8**

Table 8.6: F-scores on WSJ section 24 of reranker-best parses where the distribution-features were either the parser baseline score, the candidate's pairwise F-score, or the candidates pairwise D-score. The feature weight for these distribution-features was either learned using the default method (Scaling Factor), or empirically optimized on heldout data (Emp Optim).

8.4.3 Quantization

Probability distribution features are typically real-valued numbers, rather than integer-valued or indicator-valued numbers as is the case for most other features in a reranking model. In the previous section we discussed learning a scaling factor (either during reranker training or by empirical optimization) for the feature. In this section we will discuss another approach, from machine learning; optimizing continuous-valued features by quantization, i.e., grouping the feature values into a pre-defined number of bins, then treating each bin as an individual (indicator) feature, such as the approach by Bernal et al. [19] to modeling exon length.

One possible problem with learning a single scaling factor for a continuous-valued feature like our distribution features is that the scaling factor represents a linear weighting function. This linear function, in conjunction with the requirement that the reranking model down-weight the baseline score, can result in over-weighting candidates low in the n-best list. In order to address the problem that a linear function may not be well-suited for weighting a real-valued feature such as candidate scores, several options for *quantizing* the score were considered. Quantization consists of grouping the possible values of a variable from a continuous set (such as real numbers) to a discrete set (such as integers). With a discrete set of values and by learning a separate weight for each possible values, in essence one can learn a non-linear function for the baseline system–score feature. Note that distributing the baseline score into bins is not a novel concept, and was in fact explored as an option in the original experimental setup by Charniak and Johnson [44], who found it to make little difference. These findings were replicated here, but as will be demonstrated, under different conditions it does prove beneficial to bin the baseline-score feature.

The first method explored here for quantizing the baseline-score is to simply define a set of *bins* and assign each candidate to a bin according to the distribution-score for that candidate. Here, this straightforward quantization is complicated by the fact that the log probability scores output by the Charniak baseline parser are not normalized, and thus are not comparable across different sentences. The probability scores are also affected by sentence length, such that the one-best candidates for short sentences have greater probability according to the parsing model than the one-best candidates for long sentences. Thus, the log probabilities output by the parser were conditionally normalized before quantizing into bins.

Another method for quantizing the baseline-score is to use the rank of a candidate as its score. The benefit of using the rank as a feature is that rank is comparable across sentences, thus there is no need for normalization. In this section the rank of a candidate is treated as a binned feature, i.e., with a separate feature defined for each bin. There are several options for defining the boundaries of the bins for such a feature, including linearly dividing the ranks into equally-sized bins, and exponentially dividing the ranks such that the lower ranks are placed into larger bins and higher ranks are spread more sparsely across the bins. Herein we investigate the straightforward method of defining a separate bin for each rank value.

One could also create a scalar feature to represent the rank values, i.e., by defining a single feature for the rank score, and multiplying the learned weight of the feature by the rank value of the candidate. Although this option was explored briefly, there were similar problems to using a distribution score as a scalar feature, namely that a linear function is inadequate for weighting such a feature.

Table 8.7 presents the results of these two methods of quantizing the parser probability score to train a reranking model. Again, the Parser-Best column shows the F-score accuracy of the baseline-best parse candidate, and the Scaling Factor column shows the F-score accuracy of the reranker-best parse candidate using the de-facto standard method of learning a feature weight for the baseline score within the reranking model.

The two methods for quantizing the parser-probability distribution feature improve over the

Condition	Baseline	Scaling Factor	Binned CLP	Binned Rank
50-best	88.9	90.2	90.4	90.2
Self-Trained	90.2	90.9	91.1	90.8
50-best ∪ CSLUt	89.4	90.3	90.6	90.5

Table 8.7: F-scores on WSJ section 24 of reranker-best parses where the weight for the baseline-score feature was quantized into bins by Conditional Log-Probability (CLP), or Rank.

scaling factor method under some circumstances. By normalizing the log-probability scores and binning, the F-score of the reranker output improves for three of our test conditions: 50-best and Self-trained improve very slightly, with a 0.2 increase in F-score above the default method; the 50-best ∪ CSLUt-constrained results show a slightly larger improvement with a 0.3 absolute increase in F-score. Using the rank of each candidate as a binned feature is less effective, providing no increase in F-score under the 50-best condition, a smaller improvement of 0.2 F-score for 50-best ∪ CSLUt-constrained, and a marginal decrease in F-score for Self-Trained, from 90.9 to 90.8.

Application to Alternate Distributions

Table 8.8 shows the results of using the Pairwise-F and Pairwise-D metrics as quantized features in a reranking model. Surprisingly, our method of normalizing the score then quantizing into bins (Binned CLP in the table) is actually detrimental when using either of the pairwise metrics in comparison to using the parser probability score. We suspect that the pairwise scores were not equally spread across the defined bins; candidates in an n-best list tend to be highly similar as we showed in Chapter 7 and thus do not follow a normal distribution, which was not taken into account when the bin boundaries were defined.

The reranking models trained with the binned pairwise rank of the candidate as a feature performed well, which is unsurprising given the improved rank-order accuracy of the pairwise metrics (shown in Chapter 7, p. 104). Both the 50-best and Self-trained outputs showed an increase in reranker-best F-score when using either of the two pairwise metrics to rank the candidates in comparison to using the candidate rank derived from the parser probability score. The 50-best ∪ CSLUt-constrained condition provides a small exception, showing a slight decrease in F-score when using the pairwise metrics rather than the parser probability score.

We also experimented with learning a reranking model with all three scoring metrics (parser-probability, Pairwise-F, and Pairwise-D), using each of the different methods for learning with distribution scores, but ultimately we did not see a noticeable difference from using just one of the scores as a feature, perhaps indicating that the different scoring metrics do not provide complementary information in a reranking model.

Interestingly, there was not a clear winner in terms of the "best" feature-weighting method

	Pairwise-F			Pairwise-D		
Condition	Scaling Factor	Binned CLP	Binned Rank	Scaling Factor	Binned CLP	Binned Rank
50-best	90.1	89.6	90.6	90.5	89.9	90.6
Self-trained	90.5	90.1	90.8	90.9	90.6	90.9
50-best ∪ CSLUt	90.1	89.8	90.3	90.4	89.9	90.4

Table 8.8: F-scores on WSJ section 24 of reranker-best parses where the distribution features were either the parser baseline score, the candidate's pairwise F-score, or the candidates pairwise D-score. The feature weight for these distribution features was either learned using the default method (Scaling Factor), or using one of two methods to quantize the distribution feature: Binned Conditional Log-Probability (CLP), or Binned Rank.

and metric. The 50-best condition performed best using the Binned Rank function on either the Pairwise-F score or the Pairwise-D score; 50-best $\cup$ CSLUt performed best with empirical optimization on the Pairwise-D score; and Self-Training performed best using the Binned Conditional Log-Probability on the parser probability score. The conclusion to be drawn from these results is that, since the effectiveness of using any function to derive reranking features from any baseline score will vary across different datasets, best-practices for reranking might do well to consider a range of functions and scores rather than arbitrarily using the current de-facto standards.

In this section we have shown that different levels of reranker performance can be obtained through different functions for defining, optimizing, and representing a probability distribution as a feature in the reranking model. The methods presented for deriving distributions over the candidate and learning feature-weights for these distribution features could be considered alongside other current-best practices for training reranking models.

8.5 Conclusion

In this chapter we provided several methods for characterizing the distribution over weighted constraints. We introduced *kurtosis* as a formal calculation of a distribution's peakedness, and this formal metric confirmed the empirical estimation of peakedness that results from calculating the normalized score of the top-weighted constraints. By comparing different distributions using the Kullback-Leibler divergence statistic, we were able to determine when one distribution results in a large *information gain* over another, which was particularly evident under the Self-training condition. We confirmed empirically that removing the distribution feature from a downstream reranker is detrimental to performance, though to somewhat varying degrees for each of our reported conditions. Therefore, we hypothesized that including the distribution feature in training might result in weight under-training for the other features, similar to the effects noted by Sutton et al. [203]. In the final sections we discussed a few different methods for optimizing a distribution feature. The best performance gains were on the 50-best $\cup$ CSLUt-constrained condition, by *empirically optimizing* either the parser probability score or the Pairwise-D distance metric (from Chapter 7). Other gains were also observed, though to a lesser degree, for other conditions using these optimization methods.

In conclusion, the distribution over a constrained space can be characterized quantitatively. Features based on such a distribution clearly impact performance, and the method used for optimizing these features also impacts performance. Future researchers would do well to experiment with different distributions and optimization methods.

Chapter 9

Conclusion

The goal of this research was to enable the general comparison, analysis, and improvement of pipeline systems based on the use and characteristics of constraints in the pipeline. To achieve this goal, we defined a formal framework to generalize pipeline systems, and validated that framework by creating a classification system for existing pipeline systems. We also defined a set of metrics to quantitatively measure specific characteristics of a constrained space, including the diversity, regularity, density, and peakedness of the space. Our long series of experiments, manipulating the use and characteristics of constraints in two different parsing pipelines, provided empirical evidence of how constraint characteristics affect pipeline performance. Several of our experiments showed that the unexpected underperformance of some existing pipeline improvement techniques was due to disregarding the impact of constraint characteristics on pipeline performance. By defining and utilizing new metrics to characterize the constraints used in a pipeline, this dissertation has provided several methods to better understand how constraints affect pipeline performance and how best to alter the stages of and constraints within a pipeline to improve performance.

9.1 Chapter Summaries

In Chapter 2, we defined a formal framework of pipeline systems, with four main elements: message passing, constraint representation, stage-internal attributes, and pipeline evaluation. We discussed three methods of passing messages in a pipeline, namely feed-forward, feedback, and iteration. We also discussed different sources of the messages, including single-source versus multi-source, sampling across a pipeline, and external manipulation of the messages. This framework allowed us to address questions on the effects on pipeline performance of different representations of constraints, different constraint sources, and the method by which constraints are passed through the pipeline.

Chapter 3 used our pipeline framework to classify a large set of existing pipelines from parsing to machine translation to automatic speech recognition to image recognition. The systems were classified based on constraint representation, constraint-source types, and overall structure of the pipeline. Such a large-scale classification was used to validate the formal framework, demonstrating that it is sufficient to cover any existing pipeline system. This large-scale classification may serve as a benchmark for future researchers to classify their pipeline systems, as well as to assist in determining the best type of pipeline system for different research problems.

Our novel technique of *pipeline iteration* was proposed and examined in detail in Chapter 4, along with the effects of iteration on pipeline performance. In this chapter we also discussed ways in which to identify the two different types of errors in pipeline systems (*search errors* and *model errors*), what causes the errors, and a few ways to recover from them. We empirically determined that iterated constraints typically performed better than non-iterated constraints. Using a precision/recall tradeoff technique, we systematically varied the restrictiveness of a constraint set, and showed that heavily-constrained spaces outperformed the less-constrained spaces defined by using only high-confidence (high-precision) constraints. We showed a 0.6% absolute F-score improvement over the hard baseline defined by the Charniak and Johnson [44] pipeline, using the same pipeline. In taking the union of n-best lists, we were able to analyze the search errors and model errors occurring in the pipeline, and determine that improvements can be made to pipeline accuracy using constraints to resolve model errors.

Chapter 5 examined interactions between models in a pipeline system. We demonstrated that strictly enforcing the ordering of model complexity in a pipeline can hurt pipeline accuracy, and with our combined system produced new state-of-the-art results for NP-Chunking and shallow parsing, higher than any previously reported result by 0.5% absolute F-score. We saw that allowing a mismatch between train- and test-time conditions can result in the highest levels of performance; we also saw a few cases where a "mis-trained" reranking model outperformed the trained model in certain cases.

In Chapter 6, we showed that the commonly-reported metrics of one-best and oracle-best candidates are insufficient predictors of downstream pipeline performance.

Chapter 7 captured the spatial characteristics of a set of constraints by presenting quantitative metrics based on graph-theory to measure the *diversity, regularity, density,* and *coverage* of a constrained space. Diversity has been mentioned often in the pipeline literature but never formally defined or measured, while the other characteristics have been disregarded in the literature. A simple method for manipulating the diversity of a constraint set quantifiably affected the density of the manipulated sets, indicating that the density metric may also be used to formally measure diversity. We also examined a novel method for generating n-best constraints. Using the Pairwise-D distance metric defined in this chapter, we were able to improve the rank-accuracy of n-best parse lists output by the Charniak/Johnson parser and reranker. By defining formal metrics to measure the area and density of a space, this chapter has opened the door for quantifiable comparisons across different data sets, and indeed, across different application areas.

In Chapter 8, we provided several methods for characterizing a distribution over weighted constraints. We introduced kurtosis as a quantitative measurement of a distribution's *peakedness*. We compared different distributions using the Kullback-Leibler divergence statistic to determine when a particular distribution resulted in a large *information gain* over another. We confirmed empirically that removing the probability-score feature from a downstream reranker was detrimental to performance, and discussed a few novel methods for optimizing a probability feature. Using empirical optimization to train the parameter weight of probability distribution features, we achieved a significant gain under some conditions, up to 0.6% absolute F-score. Results in this chapter clearly demonstrated that distribution-based features—and the method used for optimizing these features—impact pipeline performance.

In summary: having established the general nature of pipelines in NLP in Chapter 1, in Chapter 2 we constructed a framework of pipeline systems, which was used in Chapter 3 to conduct a large-scale survey of existing pipeline systems and classify these systems according to the pipeline framework. Chapters 4 and 5 established several general techniques for pipeline improvement that achieved significant improvements over state-of-the-art context-free parsing results, while Chapters 6–8 analyzed several characteristics of constraints in pipelines, and demonstrated how altering those characteristics affected pipeline performance.

9.2 Best Practices

As stated at the beginning of this thesis, one of its aims was to better understand, analyze, and evaluate pipeline systems in order to discover a set of best practices for working with pipelines. This section presents a summary of those best practices.

Design with the downstream stage in mind.

Be willing to tailor the output of each stage to optimize for downstream processes. Traditionally, each stage in a pipeline is designed independently of other stages in the pipeline, in order to retain the *modularity* that is one of the benefits of working with a pipeline system. This best practice may decrease the modularity of a pipeline stage, but this effect will likely be offset by an improvement in pipeline performance.

In the majority of the experiments presented in the preceding chapters, the initial stage was optimized for a one-best (maximum likelihood) objective, followed by a reranking stage. Some of the experiments (Chapters 4, 6, 8) were designed to address (or work around) this mismatch between a model optimized for a single-best solution output and its downstream stage which by definition required multiple solutions as input. Clearly, there is interaction between the stages in a pipeline, and designing for that interaction, rather than ignoring it in favor of stage modularity, may prove beneficial for many pipeline systems.

Another aspect of designing for the downstream stage can be seen in our work on cell-closure pipelines [181, 182]. By taking into account the search algorithm to be used in the downstream parser, we were able to specifically design constraints to effect efficiency improvements while also retaining accuracy. The more traditional route of generating partial solutions (such as NP chunks or shallow chunks) to constrain a downstream parser have not been as effective at improving pipeline performance.

This best practice does *not* mean that pipeline stages cannot or should not function as independent systems, just that a pair of stages specifically designed to interact in a pipeline may outperform two independently designed stages glued together to create a pipeline.

Consider characteristics of the space.

When generating an n-best list (or lattice, or forest), consider characteristics *other than* the accuracy of the top-ranked candidate. Chapters 7 and 8 were devoted to quantifying various characteristics of a constrained space, and demonstrating the effects of those characteristics on pipeline performance.

In Chapter 6, we demonstrated that mid-pipeline *accuracy* (one-best and oracle-best) did not predict pipeline-final accuracy; in Chapter 7, we verified this fact, demonstrating that we could achieve high-accuracy pipeline-final results even with low oracle-best rates. That same set of experiments also demonstrated that the *size* of the space did not predict performance. *Coverage* of the space was the best performance predictor, and generating a fully-covered space using our simple tree-transformation algorithm was the most effective at improving the performance of our pipeline-final reranker. *Peakedness* of the search space was a poor predictor but greatly influenced feature-weight training in the reranker model, frequently causing weight undertraining of other features in the model.

This best practice is related to designing with the downstream in mind. For example, we showed in Chapter 7 that diversity in an n-best list does not work well for reranking, even though Sagae and Lavie [189] clearly demonstrated that diversity is beneficial for recombination stages downstream. Thus the downstream stage will affect which characteristics of the space are relevant and helpful or harmful.

Utilize "free" information.

Learn to recognize, extract, and implement free information when it is available. Let us define free information as that which is available with no additional man-hours devoted to creating a new resource (either data or models). In Chapter 4, we extracted information from our pipeline-final output and used that information to constrain the pipeline in a second pass. That information resulted in a slight improvement on a hard baseline, but more importantly, we achieved that improvement without designing new features or models. Thus the improvement was available "for free."

This best practice is very similar to the self-training work by McClosky et al. [152, 153], where they used existing models to parse new data (unannotated corpora), then re-trained the parser model in a second pass. Here they used models trained on annotated corpora to generate more data, which they treated as additional training data, and achieved an improvement in their pipeline performance without requiring additional (expensive) annotated data.

Consider the different dimensions of a pipeline system.

Finally, recognize that there are several dimensions for experimentation when working with pipeline systems: the design of the pipeline itself; the constraint-types passed downstream from an upstream stage; and the model features, search algorithm, and optimization objective at each stage in the pipeline. Experiment with *each* of these dimensions to find the parameters that work best for your pipeline.

9.3 Future Work

There are many future directions for this research. While the empirical results presented in this dissertation were focused on parsing, the techniques shown here are by no means limited to parsing pipelines, and could easily be applied to pipeline systems in other fields. One obvious future direction will be to apply the techniques of pipeline iteration, train/test mismatching, and n-best constraint generation to other NLP pipelines, including machine translation (MT), speech recognition, speech synthesis, and language analysis pipelines.

For example, pipeline iteration (Chapter 4) could be applied to MT by extracting high-precision partial translations from the output of a first pass through the translation decoder, then used to accurately reduce the search space of a second pass through the decoder. One could also extract high-agreement translation boundaries and similarly constrain a second decoding pass; lexical re-ordering could be another such constraint on the MT search space. There are numerous possible applications of our pipeline iteration technique for MT systems.

Predicting parse-reranking underperformance and methods to mitigate such underperformance effects has been a major focus of this dissertation. Reranking translation candidates has received some attention Kumar et al. [136], Shen et al. [199], but results have not been positive. It would be very interesting to explore whether similar methods as those presented in this dissertation for improving parse-reranking would also be effective for improving machine translation hypothesis reranking.

Our work on chart-cell constraints [181, 182] has inspired some efficiency improvements in an MT system Xiong et al. [213]; it has also been discussed as a generic pruning technique for efficiency improvements Hopkins and Langmead [111]. Clearly these techniques are not limited to just parsing pipelines and indeed are not even limited to pipeline systems.

This dissertation has shown that pipelines are typically implemented to address scalability problems for particular applications, and as increased computational power becomes available over the years, the pipeline architecture is no longer needed to address scalability for those same applications. However, new applications will arise which are beyond the scope of today's computational power; it would be interesting to see how well the framework and techniques presented in this dissertation retain their applicability as these new applications appear. Future research addressing problems of scalability will be hugely impacted by the advent of cloud computing, and exploring the similarities between the pipeline architecture and the cloud architecture could result in some interesting directions for future research.

Bibliography

[1] Steven Abney. 1991. Parsing by chunks. In Robert Berwick, Steven Abney, and Carol Tenny, editors, *Principle-Based Parsing*. Kluwer Academic Publishers, pages 257–278.

[2] Steven Abney. 1996. Partial parsing via finite-state cascades. *Journal of Natural Language Engineering*, 2(4):337–344.

[3] Hiyan Alshawi, Shona Douglas, and Srinivas Bangalore. 2000. Learning dependency translation models as collections of finite-state head transducers. *Computational Linguistics*, 26:45–60.

[4] Rie Kubota Ando and Tong Zhang. 2005. A high-performance semi-supervised learning method for text chunking. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 1–9.

[5] Abhishek Arun, Chris Dyer, Barry Haddow, Phil Blunsom, Adam Lopez, and Philipp Koehn. 2009. Monte Carlo inference and maximization for phrase-based translation. In *Proceedings of the 13th Conference on Computational Natural Language Learning (CoNLL)*, pages 102–110.

[6] Vassilis Athitsos, Jonathan Alon, and Stan Sclaroff. 2005. Efficient nearest neighbor classification using a cascade of approximate similarity measures. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 486–493.

[7] Necip Fazil Ayan and Bonnie J. Dorr. 2006. Going beyond AER: an extensive analysis of word alignments and their impact on MT. In *Proceedings of the 21st International Conference on Computational Linguistics (COLING) and the 44th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 9–16.

[8] Michiel Bacchiani, Michael Riley, Brian Roark, and Richard Sproat. 2006. MAP adaptation of stochastic grammars. *Computer Speech and Language*, 20:41–68.

[9] Michiel Bacchiani, Brian Roark, and Murat Saraclar. 2004. Language model adaptation with MAP estimation and the perceptron algorithm. In *Proceedings of the HLT-NAACL Annual Meeting*, pages 21–24.

[10] James K. Baker. 1979. Trainable grammars for speech recognition. In *Proceedings of the 97th Meeting of the Acoustical Society of America*, pages 547–550.

[11] Jean-Marie Balfourier, Philippe Blache, and Tristan van Rullen. 2002. From shallow to deep parsing using constraint satisfaction. In *Proceedings of the 19th International Conference on Computational Linguistics (COLING)*, pages 36–42.

[12] Srinivas Bangalore. 1996. "Almost parsing" technique for language modeling. In *Proceedings of the 4th International Conference on Speech and Language Processing (ICSLP)*, pages 1173–1176.

[13] Srinivas Bangalore and Aravind K. Joshi. 1999. Supertagging: An approach to almost parsing. *Computational Linguistics*, 25(2):237–265.

[14] Srinivas Bangalore, Vanessa Murdock, and Giuseppe Riccardi. 2002. Bootstrapping bilingual data using consensus translation for a multilingual instant messaging system. In *Proceedings of the 19th International Conference in Computational Linguistics (COLING)*.

[15] Srinivas Bangalore and Giuseppe Riccardi. 2000. Finite-state models for lexical reordering in spoken language translation. In *Proceedings of the 6th International Conference on Speech and Language Processing (ICSLP)*.

[16] Srinivas Bangalore and Giuseppe Riccardi. 2001. A finite-state approach to machine translation. In *Proceedings of the 2nd Annual Meeting of the North American Association of Computational Linguistics (NAACL)*.

[17] Colin Bannard and Chris Callison-Burch. 2005. Paraphrasing with bilingual parallel corpora. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL)*.

[18] F. Bergholm. 1987. Edge focusing. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 9(6):726–741.

[19] Axel Bernal, Koby Crammer, Artemis Hatzigeorgiou, and Fernando C.N. Pereira. 2007. Global discriminative learning for higher-accuracy computational gene prediction. *Public Library of Science Computational Biology*, 3(3):0488–0497.

[20] Daniel M. Bikel. 2004. Intricacies of Collins' parsing model. *Computational Linguistics*, 30(4):479–511.

[21] Jeff Bilmes. 1997. A gentle tutorial on the EM algorithm and its application to parameter estimation for Gaussian mixture and hidden Markov models. Technical Report ICSI-TR-97-021, International Computer Science Institute.

[22] Alexandra Birch, Chris Callison-Burch, and Miles Osborne. 2006. Constraining the phrase-based, joint probability statistical translation model. In *Proceedings of the 7th Conference of the Association for Machine Translation in the Americas (AMTA)*, pages 10–18.

[23] Alexandra Birch, Miles Osborne, and Philipp Koehn. 2007. CCG supertags in factored statistical machine translation. In *Proceedings of the Second Workshop on Statistical Machine Translation at the 45th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 9–16.

[24] E. Black, S. Abney, D. Flickenger, C. Gdaniec, R. Grishman, P. Harrison, D. Hindle, R. Ingria, F. Jelinek, J. Klavans, M. Liberman, M. Marcus, S. Roukos, B. Santorini, and T. Strzalkowski. 1991. A procedure for quantitatively comparing the syntactic coverage of English grammars. In *Proceedings of the 29th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 306–311.

[25] Don Blaheta. 2004. *Function Tagging*. Ph.D. thesis, Brown University.

[26] Don Blaheta and Eugene Charniak. 1999. Automatic compensation for parser figure-of-merit flaws. In *Proceedings of the 37th Annual Meeting of ACL*, pages 513–518.

[27] Eric Brill. 1992. A simple rule-based part of speech tagger. In *Proceedings of the 3rd Conference on Applied Natural Language Processing (ANLP)*, pages 111–116.

[28] Eric Brill. 1995. Transformation-based error-driven learning and natural language processing: A case study in part-of-speech tagging. *Computational Linguistics*, 21(4):543–565.

[29] Peter F. Brown, John Cocke, Stephen A. Della Pietra, Vincent J. Della Pietra, Fredrick Jelinek, John D. Lafferty, Robert L. Mercer, and Paul S. Roossin. 1990. A statistical approach to machine translation. *Computational Linguistics*, 16(2).

[30] Peter F. Brown, Vincent J. Della Pietra, Stephen A. Della Pietra, and Robert L. Mercer. 1993. The mathematics of statistical machine translation: parameter estimation. *Computational Linguistics: Special Issue on Using Large Corpora: II*, 19(2):263–312.

[31] Sabine Buchholz, Jorn Veenstra, and Walter Daelemans. 1999. Cascaded grammatical relation assignment. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing and Very Large Corpora (EMNLP)*, pages 239–246.

[32] Fred Buckley and Frank Harary. 1990. *Distance in graphs*, 1st edition. Addison-Wesley.

[33] Chris Callison-Burch. 2008. Syntactic constraints on paraphrases extracted from parallel corpora. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

[34] Sharon A. Caraballo and Eugene Charniak. 1998. New figures of merit for best-first probabilistic chart parsing. *Computational Linguistics*, 24(2):275–298.

[35] Claire Cardie and Nicholas Howe. 1997. Improving minority class prediction using case-specific feature weights. In *Proceedings of the 14th International Conference on Machine Learning (ICML)*, pages 57–65.

[36] Lynn Carlson, Daniel Marcu, and Mary Ellen Okurowski. 2001. Building a discourse-tagged corpus in the framework of Rhetorical Structure Theory. In *Proceedings of the Second SIGdial Workshop on Discourse and Dialogue at the 39th Annual Meeting of the Association for Computational Linguistics (ACL)*.

[37] Xavier Carreras and Lluís Màrquez. 2003. Phrase recognition by filtering and ranking with perceptrons. In *Proceedings of the Conference on Recent Advances in Natural Language Processing (RANLP)*, pages 205–216.

[38] Xavier Carreras, Lluís Màrquez, and Jorge Castro. 2005. Filtering-ranking perceptron learning for partial parsing. *Machine Learning: Special Issue on Learning in Speech and Language Technologies*, 60(1-3):41–71.

[39] Ming-Wei Chang, Quang Do, and Dan Roth. 2006. Multilingual dependency parsing: A pipeline approach. In *Proceedings of Recent Advances in Natural Language Processing*, pages 55–78.

[40] Ming-Wei Chang, Quang Do, and Dan Roth. 2006. A pipeline framework for dependency parsing. In *Proceedings of the COLING/ACL 2006 Main Conference Poster Sessions*, pages 65–72.

[41] Eugene Charniak. 2000. A maximum-entropy-inspired parser. In *Proceedings of the 1st Annual Meeting of the North American Chapter of the Association for Computational Linguistics (NAACL) and 6th Conference on Applied Natural Language Processing*, pages 132–139.

[42] Eugene Charniak, Sharon Goldwater, and Mark Johnson. 1998. Edge-based best-first chart parsing. In *Proceedings of the 6th Workshop for Very Large Corpora*, pages 127–133.

[43] Eugene Charniak and Mark Johnson. 2001. Edit detection and parsing for transcribed speech. In *Proceedings of the 2nd Annual Meeting of the North American Chapter of the Association for Computational Linguistics (NAACL)*, pages 118–126, Pittsburgh, PA, USA.

[44] Eugene Charniak and Mark Johnson. 2005. Coarse-to-fine n-best parsing and MaxEnt discriminative reranking. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 173–180.

[45] Eugene Charniak, Mark Johnson, Micha Elsner, Joseph Austerweil, David Ellis, Isaac Haxton, Catherine Hill, R. Shrivaths, Jeremy Moore, Michael Pozar, and Theresa Vu. 2006. Multilevel coarse-to-fine PCFG parsing. In *Proceedings of the HLT-NAACL Annual Meeting*, pages 168–175.

[46] Eugene Charniak, Kevin Knight, and Kenji Yamada. 2003. Syntax-based language models for statistical machine translation. In *MT Summit IX*, pages 40–46.

[47] Stanley F. Chen and Joshua Goodman. 1996. An empirical study of smoothing techniques for language modeling. In *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 310–318.

[48] Colin Cherry and Dekang Lin. 2003. A probability model to improve word alignment. In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 88–95.

[49] Colin Cherry and Dekang Lin. 2006. Soft syntactic constraints for word alignment through discriminative training. In *Proceedings of the 21st International Conference on Computational Linguistics (COLING) and the 44th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 105–112.

[50] David Chiang. 2005. A hierarchical phrase-based model for statistical machine translation. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL)*.

[51] David Chiang. 2007. Hierarchical phrase-based translation. *Computational Linguistics*, 33:201–228.

[52] Jike Chong, Youngmin Yi, Arlo Faria, Nadathur Satish, and Kurt Keutzer. 2008. Data-parallel large vocabulary continuous speech recognition on graphics processors. In *Proceedings of the Workshop on Emerging Applications and Many-Core Architecture (EAMA)*.

[53] Kenneth Ward Church. 1988. A stochastic parts program and noun phrase parser for unrestricted text. In *Proceedings of the Second Conference on Applied Natural Language Processing (ANLP)*, pages 136–143.

[54] Philip J. Clark and Francis C. Evans. 1954. Distance to nearest neighbor as a measure of spatial relationships in populations. *Ecology*, 35(4):445–453.

[55] Stephen Clark, Ann Copestake, James R. Curran, Yue Zhang, Aurelie Herbelot, James Haggerty, Byung-Gyu Ahn, Curt Van Wyk, Jessika Roesner, Jonathan Kummerfeld, and Tim Dawborn. 2009. Large-scale syntactic processing: Parsing the web. Technical Report Final Report of the JHU CLSP Workshop, Johns Hopkins University.

[56] Stephen Clark and James Curran. 2006. Partial training for a lexicalized-grammar parser. In *Proceedings of the HLT-NAACL Annual Meeting*, pages 144–151.

[57] Stephen Clark and James Curran. 2007. Perceptron training for a wide-coverage lexicalized-grammar parser. In *Proceedings of ACL 2007 Workshop on Deep Linguistic Processing*, pages 9–16.

[58] Stephen Clark and James R. Curran. 2004. Parsing the WSJ using CCG and log-linear models. In *Proceedings of the 42nd Meeting of the Association for Computational Linguistics (ACL)*, pages 103–110.

[59] Stephen Clark and James R. Curran. 2007. Wide-coverage efficient statistical parsing with CCG and log-linear models. *Computational Linguistics*, 33(4):493–552.

[60] Stephen Clark, James R. Curran, and Miles Osborne. 2003. Bootstrapping POS-taggers using unlabelled data. In *Proceedings of the 7th Conference on Computational Natural Language Learning (CoNLL)*, pages 49–55.

[61] John Cocke and Jacob T. Schwartz. 1970. Programming languages and their compilers: Preliminary notes. Technical report, Courant Institute of Mathematical Sciences, New York University.

[62] Trevor Cohn, Andrew Smith, and Miles Osborne. 2005. Scaling conditional random fields using error-correcting codes. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL)*.

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

[64] Michael Collins. 2000. Discriminative reranking for natural language parsing. In *Proceedings of the 17th International Conference on Machine Learning (ICML)*, pages 175–182.

[65] Michael Collins. 2002. Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1–8.

[66] Michael Collins and Nigel Duffy. 2002. New ranking algorithms for parsing and tagging: Kernels over discrete structures, and the voted perceptron. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 263–270.

[67] Michael Collins, Philipp Koehn, and Ivona Kucerova. 2005. Clause restructuring for statistical machine translation. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 531–540.

[68] Michael Collins and Terry Koo. 2005. Discriminative reranking for natural language parsing. *Computational Linguistics*, 31(1):25–69.

[69] Brooke Cowan and Michael Collins. 2005. Morphology and reranking for the statistical parsing of Spanish. In *Proceedings of the Human Language Technology Conference and the Conference on Empirical Methods in Natural Language Processing (HLT/EMNLP)*.

[70] Koby Crammer and Yoram Singer. 2001. PRanking with ranking. In *Proceedings of the 14th Annual Conference on Neural Information Processing Systems (NIPS)*, pages 641–647.

[71] James R. Curran, Stephen Clark, and David Vadas. 2006. Multi-tagging for lexicalized-grammar parsing. In *Proceedings of the 21st International Conference on Computational Linguistics (COLING) and the 44th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 697–704.

[72] Sebastian van Delden and Fernando Gomez. 2004. Cascaded finite-state partial parsing: A larger-first approach. In Nicolas Nicolov, Kalina Bontcheva, Galia Angelova, and Ruslan Mitkov, editors, *Recent Advances in Natural Language Processing III, Selected Papers from RANLP 2003*, volume 260 of *Current Issues in Linguistic Theory (CILT)*. John Benjamins, pages 217–226.

[73] Sebastian van Delden and Fernando Gomez. 2006. Improving inter-level communication in cascaded finite-state partial parsers. In *Selected Papers from the Fifth International Workshop on Finite State Methods in Natural Language Processing (FSMNLP)*, volume 4006 of *Lecture Notes in Computer Science*. Springer, pages 259–270.

[74] John DeNero, Mohit Bansal, Adam Pauls, and Dan Klein. 2009. Efficient parsing for transducer grammars. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics (HLT-NAACL)*, pages 227–235.

[75] John DeNero, Alexandre Bouchard-Côté, and Dan Klein. 2008. Syntactic constraints on paraphrases extracted from parallel corpora. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 314–323.

[76] John DeNero, David Chiang, and Kevin Knight. 2009. Fast consensus decoding over translation forests. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*, pages 567–575.

[77] Thomas G. Dietterich. 2000. Ensemble methods in machine learning. In *Proceedings of the 1st International Workshop on Multiple Classifier Systems*, pages 1–15.

[78] Thomas G. Dietterich. 2002. Ensemble learning. In M.A. Arbib, editor, *The handbook of brain theory and neural networks*, 2nd edition. MIT Press, Cambridge MA, USA, pages 405–408.

[79] Bojan Djordjevic, James R. Curran, and Stephen Clark. 2007. Improving the efficiency of a wide-coverage CCG parser. In *Proceedings of the 10th International Workshop on Parsing Technologies (IWPT) at ACL*, pages 39–47.

[80] Jay Earley. 1970. An efficient context-free parsing algorithm. *Communications of the Association for Computing Machinery*, 13(2):94–102.

[81] F. J. Estrada and J. H. Elder. 2006. Multi-scale contour extraction based on natural image statistics. In *Proceedings of the Computer Vision and Pattern Recognition Workshop (CVPRW)*, page 183.

[82] Michael Fink and Pietro Perona. 2003. Mutual boosting for contextual inference. In *Proceedings of the Neural Information Processing Systems Conference (NIPS)*.

[83] Jenny Rose Finkel, Christopher D. Manning, and Andrew Y. Ng. 2006. Solving the problem of cascading errors: approximate Bayesian inference for linguistic annotation pipelines. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 618–626.

[84] Jonathan Fiscus. 1997. A post-processing system to yield reduced word error rates: Recognizer output voting error reduction (ROVER). In *Proceedings of the IEEE Workshop on Automatic Speech Recognition and Understanding*, pages 347–352.

[85] Seeger Fisher and Brian Roark. 2007. The utility of parse-derived features for automatic discourse segmentation. In *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics (ACL)*, pages 488–495.

[86] Jerry A. Fodor. 1983. *The modularity of mind.* MIT Press.

[87] Victoria Fossum and Kevin Knight. 2009. Combining constituent parsers. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics (HLT/NAACL), Companion Volume: Short Papers*, pages 253–256.

[88] Heidi J. Fox. 2005. Dependency-based statistical machine translation. In *Proceedings of the Student Workshop at the 43rd Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 91–96.

[89] W. Nelson Francis and Henry Kučera. 1982. *Frequency analysis of English usage.* Houghton Mifflin, Boston MA, USA.

[90] Annette Frank, Markus Becker, Berthold Crysmann, Bernd Kiefer, and Ulrich Schäfer. 2003. Integrated shallow and deep parsing: TopP meets HPSG. In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 104–111.

[91] Ryan Gabbard, Seth Kulick, and Mitchell Marcus. 2006. Fully parsing the Penn treebank. In *Proceedings of the HLT-NAACL Annual Meeting*, pages 184–191.

[92] William A. Gale and Kenneth W. Church. 1993. A program for aligning sentences in bilingual corpora. *Computational Linguistics*, 19(1):75–102.

[93] Jianfeng Gao, Galen Andrew, Mark Johnson, and Kristina Toutanova. 2007. A comparative study of parameter estimation methods for statistical natural language processing. In *Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 824–831.

[94] L. Gillick and Stephen Cox. 1989. Some statistical issues in the comparison of speech recognition algorithms. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pages 532–535.

[95] Elliot Glaysher and Dan Moldovan. 2006. Speeding up full syntactic parsing by leveraging partial parsing decisions. In *Proceedings of the COLING/ACL Main Conference Poster Sessions*, pages 295–300.

[96] John J. Godfrey, Edward C. Holliman, and Jane McDaniel. 1992. SWITCHBOARD: telephone speech corpus for research and development. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pages 517–520.

[97] Vaibhava Goel, Shankar Kumar, and William Byrne. 2000. Segmental minimum Bayes-risk ASR voting strategies. In *Proceedings of the International Conference on Spoken Language Processing (ICSLP)*, pages 139–142.

[98] Joshua Goodman. 1996. Parsing algorithms and metrics. In *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 177–183.

[99] Joshua Goodman. 1997. Global thresholding and multiple-pass parsing. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 11–25.

[100] Joshua Goodman. 1998. *Parsing inside-out.* Ph.D. thesis, Harvard University.

[101] Jan Hajič, Alena Böhmová, Eva Hajičová, and Barbora Vidová-Hladká. 2000. The Prague dependency treebank: a three-level annotation scenario. In A. Abeillé, editor, *Treebanks: Building and Using Parsed Corpora.* Kluwer, Amsterdam, pages 103–127.

[102] Keith Hall. 2004. *Best-first word-lattice parsing: Techniques for integrated syntactic language modeling.* Ph.D. thesis, Brown University.

[103] Keith Hall, Jiri Havelka, and David A. Smith. 2007. Log-linear models of non-projective trees, k-best MST parsing and tree-ranking. In *Proceedings of the CoNLL Shared Task Session of EMNLP-CoNLL 2007*, pages 962–966.

[104] William Rowan Hamilton. 1856. Memorandum respecting a new system of roots of unity. *Philosophical Magazine*, 12:446.

[105] Richard W. Hamming. 1950. Error detecting and error correcting codes. *Bell System Technical Journal*, 26(2):147–160.

[106] Peter E. Hart. 1968. The condensed nearest neighbor rule. *IEEE Transactions on Information Theory*, IT(14):515–516.

[107] Hany Hassan, Khalil Sima'an, and Andy Way. 2007. Supertagged phrase-based statistical machine translation. In *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics (ACL)*, pages 288–295.

[108] John C. Henderson and Eric Brill. 1999. Exploiting diversity in natural language processing: Combining parsers. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing and Very Large Corpora (EMNLP)*, pages 187–194.

[109] Kristy Hollingshead, Seeger Fisher, and Brian Roark. 2005. Comparing and combining finite-state and context-free parsers. In *Proceedings of the Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing (HLT/EMNLP)*, pages 787–794.

[110] Kristy Hollingshead and Brian Roark. 2007. Pipeline iteration. In *Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 952–959.

[111] Mark Hopkins and Greg Langmead. 2009. Cube pruning as heuristic search. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 62–71.

[112] Liang Huang and David Chiang. 2005. Better k-best parsing. In *Proceedings of the 9th International Workshop on Parsing Technologies (IWPT) at the Human Language Technology Conference and the Conference on Empirical Methods in Natural Language Processing (HLT/EMNLP)*, pages 53–64.

[113] Liang Huang and David Chiang. 2007. Forest rescoring: faster decoding with integrated language models. In *Proceedings of the HLT/NAACL Annual Meeting*, pages 144–151.

[114] Xuedong Huang, Alex Acero, and Hsiao-Wuen Hon. 2001. *Spoken language processing: A guide to theory, algorithm and system development.* Prentice-Hall. Cited by Chongetal08 for beam search in ASR systems.

[115] Martin Jansche. 2005. Algorithms for minimum risk chunking. In *Proceedings of the 5th International Workshop on Finite-State Methods in Natural Language Processing (FSMNLP)*, pages 97–109, Helsinki, Finland.

[116] Mark Johnson, Stuart Geman, Stephen Canon, Zhiyi Chi, and Stefan Riezler. 1999. Estimators for stochastic "Unification-Based" grammars. In *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 535–541.

[117] Aravind K. Joshi and Srinivas Bangalore. 1994. Disambiguation of super parts of speech (or supertags): Almost parsing. In *Proceedings of the 15th International Conference on Computational Linguistics (COLING)*, pages 154–160.

[118] Aravind K. Joshi, Leon S. Levy, and Masako Takahashi. 1975. Tree adjunct grammars. *Journal of Computer and System Sciences*, 10(1):136–163.

[119] Aravind K. Joshi and Yves Schabes. 1997. Tree-adjoining grammars. In A. Salomma and G. Rosenberg, editors, *Handbook of Formal Languages and Automata*, volume 3. Springer-Verlag, pages 69–123.

[120] Daniel Jurafsky and James H. Martin. 2000. *Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition*, 1st edition. Prentice-Hall.

[121] Lauri Karttunen. 1998. The proper treatment of optimality in computational phonology. In *Proceedings of Finite-State Methods for Natural Language Processing (FSMNLP)*.

[122] Tadao Kasami. 1965. An efficient recognition and syntax-analysis algorithm for context-free languages. Technical Report Scientific report AFCRL-65-758, Air Force Cambridge Research Lab, Bedford MA, USA.

[123] Martin Kay. 1984. Functional unification grammar: a formalism for machine translation. In *Proceedings of the 10th International Conference on Computational Linguistics and 22nd Annual Meeting of the Association for Computational Linguistics*, pages 75–78.

[124] Joungbum Kim, Sarah E. Schwarm, and Mari Ostendorf. 2004. Detecting structural metadata with decision trees and transformation-based learning. In *Proceedings of the HLT/NAACL Annual Meeting*, pages 137–144.

[125] Dan Klein and Christopher D. Manning. 2003. A* parsing: fast exact Viterbi parse selection. In *Proceedings of the HLT/NAACL Annual Meeting*, pages 40–47.

[126] Alexandre Klementiev, Dan Roth, Kevin Small, and Ivan Titov. 2009. Unsupervised rank aggregation with domain-specific expertise. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1101–1106.

[127] Kevin Knight and Yaser Al-Onaizan. 1998. Translation with finite-state devices. In *Proceedings of the 3rd Conference of the Association for Machine Translation in the Americas (AMTA)*, pages 421–437.

[128] Philipp Koehn. 2004. Pharaoh: A beam search decoder for phrase-based statistical machine translation models. In *Proceedings of the 6th Conference of the Association for Machine Translation in the Americas (AMTA)*, pages 115–124.

[129] Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, Chris Dyer, Ondrej Bojar, Alexandra Constantin, and Evan Herbst. 2007. Moses: Open source toolkit for statistical machine translation. In *Proceedings of the ACL 2007 Demonstrations Session*, pages 177–180.

[130] Phillip Koehn. 2005. Europarl: a parallel corpus for statistical machine translation. In *Proceedings of MT Summit X*.

[131] Scott Konishi, Alan Yuille, and James Coughlan. 2003. A statistical approach to multi-scale edge detection. *Image and Vision Computing*, 21(1):37–48.

[132] Taku Kudo and Yuji Matsumoto. 2001. Chunking with support vector machines. In *Proceedings of the 2nd Annual Meeting of the North American Chapter of the Association for Computational Linguistics (NAACL)*, pages 1–8.

[133] Shankar Kumar and William Byrne. 2002. Minimum Bayes-risk word alignments of bilingual texts. In *Proceedings of the 2002 Conference on Empirical Methods in Natural Language Processing*, pages 140–147.

[134] Shankar Kumar and William Byrne. 2003. A weighted finite state transducer translation template model for statistical machine translation. In *Proceedings of the HLT/NAACL Annual Meeting*, pages 63–70.

[135] Shankar Kumar and William Byrne. 2004. Minimum Bayes-risk decoding for statistical machine translation. In *Proceedings of the HLT/NAACL Annual Meeting*.

[136] Shankar Kumar, Yonggang Deng, and William Byrne. 2006. A weighted finite state transducer translation template model for statistical machine translation. *Natural Language Engineering*, 12(1):35–75.

[137] John Lafferty, Andrew McCallum, and Fernando Pereira. 2001. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of the 18th International Conference on Machine Learning (ICML)*, pages 282–289.

[138] Karim Lari and Steve J. Young. 1990. The estimation of stochastic context-free grammars using the Inside-Outside algorithm. *Computer Speech and Language*, 4(1):35–56.

[139] Akinobu Lee, Tatsuya Kawahara, and Kiyohiro Shikano. 2001. Julius — an open source real-time large vocabulary recognition engine. In *Proceedings of the 7th European Conference on Speech Communication and Technology (EUROSPEECH)*, pages 1691–1694.

[140] Kai-Fu Lee, Hsiao-Wuen Hon, and Raj Reddy. 1990. An overview of the SPHINX speech recognition system. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 38(1):35–45.

[141] Maider Lehr and Izhak Shafran. 2010. Discriminatively estimated joint acoustic, duration and language model for speech recognition. In *Proceedings of (ICASSP)*.

[142] Vladimir Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710.

[143] Ming Li, Xin Chen, Xin Li, Bin Ma, and Paul Vitànyi. 2004. The similarity metric. *IEEE Transactions on Information Theory*, 50(12):3250–3264.

[144] Xin Li and Dan Roth. 2001. Exploring evidence for shallow parsing. In *Proceedings of the 5th Conference on Computational Natural Language Learning (CoNLL)*, pages 107–110.

[145] Percy Liang, Ben Taskar, and Dan Klein. 2006. Alignment by agreement. In *Proceedings of the HLT/NAACL Annual Meeting*, pages 104–111.

[146] Dekang Lin and Colin Cherry. 2003. Word alignment with cohesion constraint. In *Proceedings of the HLT/NAACL Annual Meeting*, pages 49–51.

[147] Siwei Lu and Anthony Szeto. 1993. Hierarchical artificial neural networks for edge enhancement. *Pattern Recognition*, 26(8):1149–1163.

[148] Christopher D. Manning and Heinrich Schütze. 1999. *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge MA, USA.

[149] Daniel Marcu and William Wong. 2002. A phrase-based, joint probability model for statistical machine translation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 133–139.

[150] Mitchell P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. 1993. Building a large annotated corpus of English: The Penn treebank. *Computational Linguistics*, 19(2):313–330.

[151] Johnathan May and Kevin Knight. 2006. A better n-best list: practical determinization of weighted finite tree automata. In *Proceedings of the HLT/NAACL Annual Meeting*, pages 351–358.

[152] David McClosky, Eugene Charniak, and Mark Johnson. 2006. Effective self-training for parsing. In *Proceedings of the HLT-NAACL Annual Meeting*, pages 152–159.

[153] David McClosky, Eugene Charniak, and Mark Johnson. 2006. Reranking and self-training for parser adaptation. In *Proceedings of the 44th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 337–344.

[154] Ryan McDonald, Koby Crammer, and Fernando Pereira. 2005. Flexible text segmentation with structured multilabel classification. In *Proceedings of the Human Language Technology Conference and the Conference on Empirical Methods in Natural Language Processing (HLT/EMNLP)*, pages 987–994.

[155] I. Dan Melamed. 2000. Models of translational equivalence among words. *Computational Linguistics*, 26(2):221–249.

[156] D. Mintz. 1994. Robust consensus based edge-detection. *Computer Vision, Graphics, and Image Processing (CVGIP): Image Understanding*, 59(2):137–153.

[157] Tom Mitchell. 1997. *Machine learning*, 1st edition. McGraw Hill.

[158] Mehryar Mohri. 1997. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23.

[159] Mehryar Mohri and Michael Riley. 2002. An efficient algorithm for the n-best-strings problem. In *Proceedings of the 7th International Conference on Spoken Language Processing (ICSLP)*, pages 1313–1315.

[160] Mehryar Mohri and Brian Roark. 2006. Probabilistic context-free grammar induction based on structural zeros. In *Proceedings of the Human Language Technology Conference of the NAACL*, pages 312–319.

[161] Christian Monson. 2009. *ParaMor: From paradigm structure to natural language morphology induction.* Ph.D. thesis, Carnegie Mellon University.

[162] Christian Monson, Kristy Hollingshead, and Brian Roark. 2010. Simulating morphological analyzers with stochastic taggers for confidence estimation. In *Multilingual Information Access Evaluation, Vol. I, 10th Workshop of the Cross-Language Evaluation Forum (CLEF 2009), Revised Selected Papers*, Lecture Notes in Computer Science. Springer.

[163] Mark-Jan Nederhof. 2001. Approximating context-free by rational transduction for example-based MT. In *Proceedings of the Workshop on Data-Driven Machine Translation at the 39th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 25–32.

[164] Frankz Josef Och and Hermann Ney. 2003. A systematic comparison of various statistical alignment models. *Computational Linguistics*, 29(1):19–51.

[165] Franz Josef Och. 2003. Minimum error rate training for statistical machine translation. In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 160–167.

[166] Franz Josef Och and Hermann Ney. 2004. The alignment template approach to statistical machine translation. *Computational Linguistics*, 30(4):417–449.

[167] Miles Osborne and Jason Baldridge. 2004. Ensemble-based active learning for parse selection. In *Proceedings of the HLT-NAACL Annual Meeting*, pages 89–96.

[168] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL).*

[169] Fernando Pereira and Michael Riley. 1997. Speech recognition by composition of weighted finite automata. In Emmanuel Roche and Yves Schabes, editors, *Finite-State Language Processing.* MIT Press, pages 431–453.

[170] Miguel Petrere, Jr. 1985. The variance of the index (R) of aggregation of Clark and Evans. *Oecologia*, 68(1):158–159.

[171] Vasin Punyakanok and Dan Roth. 2000. The use of classifiers in sequential inference. In *Proceedings of the 13th Annual Conference on Neural Information Processing Systems (NIPS).*

[172] Lance A. Ramshaw and Mitchell P. Marcus. 1995. Text chunking using transformation-based learning. In *Proceedings of the 3rd Workshop on Very Large Corpora*, pages 82–94.

[173] Adwait Ratnaparkhi. 1996. A maximum entropy part-of-speech tagger. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 133–142.

[174] Adwait Ratnaparkhi. 1997. A linear observed time statistical parser based on maximum entropy models. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1–10.

[175] Adwait Ratnaparkhi. 1999. Learning to parse natural language with maximum entropy models. *Machine Learning*, 34(1-3):151–175.

[176] Roi Reichart and Ari Rappoport. 2007. An ensemble method for selection of high quality parses. In *Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 408–415.

[177] Alfred Rényi. 1961. On measures of information and entropy. In *Proceedings of the 4th Berkeley Symposium on Mathematics, Statistics and Probability*, pages 547–561.

[178] Alfred Rényi. 1970. *Probability theory*, 1st edition. Elsevier.

[179] John A. Rice. 1995. *Mathematical statistics and data analysis*, 2nd edition. Duxbury Press.

[180] Stefan Riezler and John T. Maxwell III. 2006. Grammatical machine translation. In *Proceedings of the HLT-NAACL Annual Meeting*, pages 248–255.

[181] Brian Roark and Kristy Hollingshead. 2008. Classifying chart cells for quadratic complexity context-free inference. In *Proceedings of the 22nd International Conference on Computational Linguistics (COLING)*, pages 745–752.

[182] Brian Roark and Kristy Hollingshead. 2009. Linear complexity context-free parsing pipelines via chart constraints. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 647–655.

[183] Brian Roark, Margaret Mitchell, and Kristy Hollingshead. 2007. Syntactic complexity measures for detecting Mild Cognitive Impairment. In *Proceedings of the ACL 2007 Workshop on Biomedical Natural Language Processing (BioNLP)*, pages 1–8.

[184] Brian Roark, Murat Saraclar, and Michael Collins. 2004. Corrective language modeling for large vocabulary ASR with the perceptron algorithm. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pages 749–752.

[185] Brian Roark, Murat Saraclar, and Michael Collins. 2007. Discriminative n-gram language modeling. *Computer Speech and Language*, 21(2):373–392.

[186] Brian Roark, Murat Saraclar, Michael J. Collins, and Mark Johnson. 2004. Discriminative language modeling with conditional random fields and the perceptron algorithm. In *Proceedings of the 42nd Meeting of the Association for Computational Linguistics (ACL)*, pages 47–54.

[187] Brian Roark and Richard Sproat. 2007. *Computational approaches to morphology and syntax*, 1st edition. Oxford University Press.

[188] Peter A. Rogerson. 2001. *Statistical methods for geography*, 2nd edition. Sage Publications Ltd.

[189] Kenji Sagae and Alon Lavie. 2006. Parser combination by reparsing. In *Proceedings of the HLT-NAACL Annual Meeting*, pages 129–132.

[190] Kenji Sagae, Yusuke Miyao, and Jun'ichi Tsujii. 2007. HPSG parsing with shallow dependency constraints. In *Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 624–631.

[191] Kenji Sagae and Jun'ichi Tsujii. 2007. Dependency parsing and domain adaptation with LR models and parser ensembles. In *Proceedings of the CoNLL 2007 Shared Task at the Joint Conferences on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL'07)*, pages 624–631.

[192] Murat Saraclar and Brian Roark. 2005. Joint discriminative language modeling and utterance classification. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pages 561–564.

[193] Charles Schafer and David Yarowsky. 2003. Statistical machine translation using coercive two-level syntactic transduction. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 9–16.

[194] Fei Sha and Fernando Pereira. 2003. Shallow parsing with conditional random fields. In *Proceedings of the HLT-NAACL Annual Meeting*, pages 134–141.

[195] Izhak Shafran and William Byrne. 2004. Task-specific minimum Bayes-risk decoding using learned edit distance. In *Proceedings of the 8th International Conference on Spoken Language Processing (ICSLP)*, pages 1945–1948.

[196] Li Shao and Hwee Tou Ng. 2004. Mining new word translations from comparable corpora. In *Proceedings of the 20th International Conference on Computational Linguistics (COLING)*, pages 618–624.

[197] Libin Shen and Aravind K. Joshi. 2003. A SNoW based supertagger with application to NP chunking. In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 505–512.

[198] Libin Shen and Aravind K. Joshi. 2005. Ranking and reranking with perceptron. *Machine Learning, Special Issue on Learning in Speech and Language Technologies*, 60(1–3):73–96.

[199] Libin Shen, Anoop Sarkar, and Franz Och. 2004. Discriminative reranking for machine translation. In *Proceedings of the HLT/NAACL Annual Meeting*, pages 177–184.

[200] Andrew Smith, Trevor Cohn, and Miles Osborne. 2005. Logarithmic opinion pools for conditional random fields. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 18–25.

[201] Caroline Sporleder and Mirella Lapata. 2005. Discourse chunking and its application to sentence compression. In *Proceedings of the Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing*, pages 257–264.

[202] Andreas Stolcke. 1995. An efficient probabilistic context-free parsing algorithm that computes prefix probabilities. *Computational Linguistics*, 21(2):165–202.

[203] Charles Sutton, Michael Sindelar, and Andrew McCallum. 2006. Reducing weight undertraining in structured discriminative learning. In *Proceedings of the HLT/NAACL Annual Meeting*, pages 89–95.

[204] C.L. Tan and S.K.K. Loh. 1993. Efficient edge detection using hierarchical structures. *Pattern Recognition*, 26(1):127–135.

[205] Kumiko Tanaka and Hideya Iwasaki. 1996. Extraction of lexical translations from non-aligned corpora. In *Proceedings of the 16th International Conference on Computational Linguistics (COLING)*, pages 580–585.

[206] Erik F. Tjong Kim Sang. 2002. Introduction to the conll-2002 shared task: Language-independent named entity recognition. In *Proceedings of the 6th Conference on Computational Natural Language Learning (CoNLL)*, pages 155–158.

[207] Erik F. Tjong Kim Sang and Sabine Buchholz. 2000. Introduction to the CoNLL-2000 shared task: Chunking. In *Proceedings of the 4th Conference on Computational Natural Language Learning (CoNLL)*, pages 127–132.

[208] Ashish Venugopal, Andreas Zollmann, and Stephan Vogel. 2007. An efficient two-pass approach to synchronous-CFG driven statistical MT. In *Proceedings of the HLT/NAACL Annual Meeting*, pages 500–507.

[209] Dimitra Vergyri. 2000. *Integration of multiple knowledge sources in speech recognition using minimum error training.* Ph.D. thesis, Johns Hopkins University.

[210] Wen Wang, Andreas Stolcke, and Mary P. Harper. 2004. The use of a linguistically motivated language model in conversational speech recognition. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pages 261–264.

[211] P.C. Woodland, J.J. Odell, V. Valtchev, and S.J. Young. 1994. Large vocabulary continuous speech recognition using HTK. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pages 749–752.

[212] Dekai Wu. 1997. Stochastic inversion transduction grammars and bilingual parsing of parallel corpora. *Computational Linguistics*, 23.

[213] Deyi Xiong, Min Zhang, and Haizhou Li. 2010. Learning translation boundaries for phrase-based decoding. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics (HLT-NAACL)*, pages 136–144.

[214] Nianwen Xue. 2003. Chinese word segmentation as character tagging. *Computational Linguistics and Chinese Language Processing*, 8(1):29–47.

[215] Alexander Yeh. 2000. More accurate tests for the statistical significance of result differences. In *Proceedings of the 18th International Conference on Computational Linguistics (COLING)*, pages 947–953.

[216] Peter N. Yianilos. 2002. Normalized forms for two common metrics. Technical report, NEC Research Institute.

[217] Daniel H. Younger. 1967. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208.

[218] Xiaojin Zhu, Andrew Goldberg, Jurgen Van Gael, and David Andrzejewski. 2007. Improving diversity in ranking using absorbing random walks. In *Proceedings of the HLT/NAACL Annual Meeting*, pages 97–104.

[219] Mark H. Zweig and Gregory Campbell. 1993. Receiver-operating characteristic (ROC) plots: a fundamental evaluation tool in clinical medicine. *Clinical Chemistry*, 39(4):561–577.

Biographical Note

Kristina M. Hollingshead was born in July 1978, in Papillion Nebraska. She received her Bachelor of Arts degree in English–Creative Writing from the University of Colorado–Boulder in May 2000. In August 2002, Kristy joined the Department of Science and Engineering at OHSU as a master's student, and transferred to the PhD program in July 2003. Her professional interests are in parsing, utilizing parsing to improve other (downstream) applications, statistical machine translation, general methods in NLP for balancing system accuracy and efficiency, and NLP evaluation metrics. Kristy was a National Science Foundation (NSF) Graduate Research Fellow from 2004–2007, and has co-authored five conference publications, two workshop papers, and one journal article.