ENCODING STRUCTURED OUTPUT VALUES

Edward Loper

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

2008

Martha Palmer
Supervisor of Dissertation

Rajeev Alur
Graduate Group Chairperson

Acknowledgements

I would like to thank my advisor, Martha Palmer, whose guidance and support, and the personal time she has invested throughout my time as a graduate student, are much appreciated. Dan Gildea has been instrumental in helping me develop and focus my dissertation research topic. I would also like to thank Mitch Marcus, Fernando Pereira, and Ben Taskar, for accepting my invitation to participate in my thesis dissertation as members of the thesis committee. Finally, I would like to thank my wife, my parents, and my two brothers for their unwavering love and support.

ABSTRACT

ENCODING STRUCTURED OUTPUT VALUES

Edward Loper

Martha Palmer

Many of the Natural Language Processing tasks that we would like to model with machine learning techniques generate structured output values, such as trees, lists, or groupings. These structured output problems can be modeled by decomposing them into a set of simpler sub-problems, with well-defined and well-constrained interdependencies between sub-problems. However, the effectiveness of this approach depends to a large degree on exactly how the problem is decomposed into sub-problems; and on how those sub-problems are divided into equivalence classes.

The notion of *output encoding* can be used to examine the effects of problem decomposition on learnability for specific tasks. These effects can be divided into two general classes: local effects and global effects. Local effects, which influence the difficulty of learning individual sub-problems, depend primarily on the coherence of the classes defined by individual output tags. Global effects, which determine the model's ability to learn long-distance dependencies, depend on the information content of the output tags.

Using a *canonical encoding* as a reference point, we can define additional encodings as reversible transformations from canonical encoded structures to a new set of encoded structures. This allows us to define a space of potential encodings (and by extension, a space of potential problem decompositions). Using search methods, we can then analyze and improve upon existing problem decompositions.

In this dissertation, I apply automatic and semi-automatic methods to the problem of finding optimal problem decompositions, in the context of five specific systems (three sequence prediction systems and two semantic role labeling systems). Additionally, I show how linear and log-linear voting can be used to combine structured prediction models that use different problem decompositions, and evaluate the effectiveness of these combined systems.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Supervised machine learning uses training examples to build a model that generalizes the mapping between an input space and an output space, allowing us to predict the correct outputs for new inputs. Many of the problems that we would like to model with machine learning techniques involve *structured* output values, such as trees, lists, or groupings. Such problems are especially common in natural language processing. For example, parsing generates a tree representing the structure of an input; chunking generates a set of non-overlapping input spans; and semantic role labelling generates a mapping between input spans and argument labels. But there are also many examples of problems with structured outputs in other domains. For example, gene intron detection generates non-overlapping input spans; and scene reconstruction generates a three dimensional model from one or more input images.

An important characteristic shared by most structured output tasks is that the number of possible output values is extremely large (or even unbounded). Typically, the number of possible output values grows exponentially with the size of the input. This contrasts with classification tasks, where there are a small fixed set of possible outputs. For classification tasks, it is common to build a separate model for each output value, describing the corresponding inputs; or to build separate discriminant functions that distinguish which inputs correspond to pairs of outputs. However,

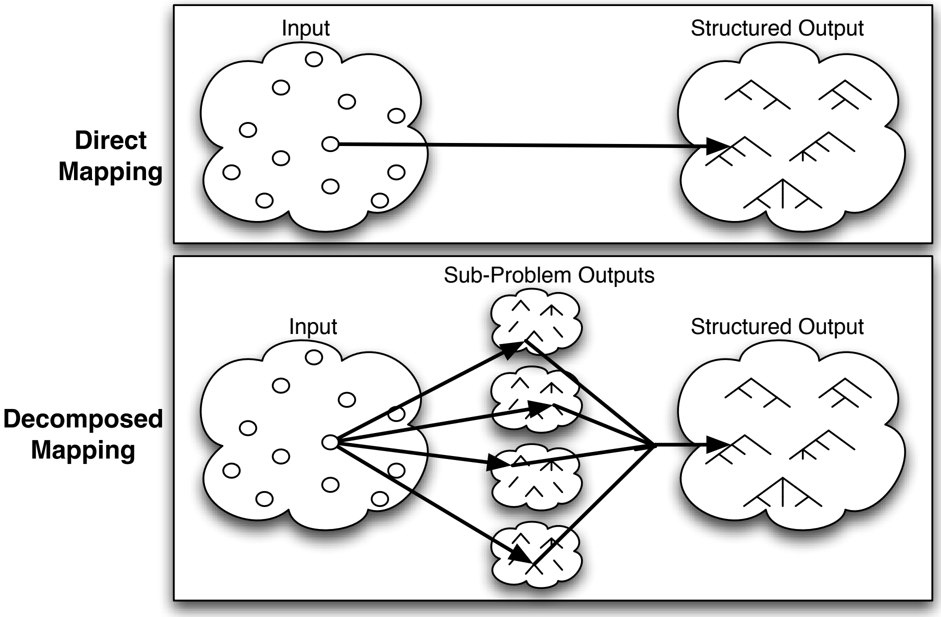


Figure 1.1: **Decomposing a Structured-Output Mapping**. In problems with structured outputs, the large number of possible output values usually makes it impractical to learn the direct mapping from inputs to outputs (top). Instead, the problem can be decomposed into a set of simpler sub-problems; and the outputs from those sub-problems can be combined to generate a structured output.

these approaches which model each output value separately are clearly impractical for structured output tasks, where the number of possible output values is often larger than the size of the training corpus.

Instead of modeling each output value separately, the problem of mapping from an input to a structured output can be decomposed into a set of simpler sub-problems, with well-defined and well-constrained interdependencies between sub-problems (Figure 1.1). Each of these sub-problems generates simple outputs, such as labels, making it possible to model them directly. In order to alleviate sparse data problems, the sub-problems are usually divided into groups of “equivalent sub-problems,” which share training data. Given an input value, the use of well-constrained interdependencies between sub-problems makes it possible to find a globally optimal solution to the sub-problems. The individual sub-problem outputs from this globally optimal solution can then be combined to generate a structured output.

The effectiveness of this approach depends to a large degree on how the problem of structured output prediction is decomposed into sub-problems; and on how those sub-problems are divided into equivalence classes. This dissertation uses *output encodings* as a tool to explore the effect of different problem decompositions on the ability of the underlying machine learning mechanism to accurately model the problem domain.

1.1 Output Encodings

An *output encoding* is an annotation scheme for structured output values, where each value is encoded as a collection of individual annotation elements. Figures 1.2–1.4 give example output encodings for various tasks. Note that there are a wide variety of possible output encodings for any output value domain.

We can use output encodings to represent problem decompositions, by establishing a one-to-one correspondence between annotation elements and sub-problems. For example, in tag-based chunking encodings such as IOB1 and IOB2, each annotation element (i.e., each tag) corresponds to a single sub-problem. The connections between the annotation elements represent the well-defined interdependencies between sub-problems. These connections are used to combine the outputs of sub-problems to generate the final structured output value. By comparing the effect of different output encodings, we can gain insight into the relationship between the corresponding problem decompositions.

In addition to specifying how the problem should be decomposed into sub-problems, we must also specify what method will be used to find the best overall solution for a given input value. Many techniques have been developed for globally optimizing various subproblem decomposition types, such as linear chains or tree structures. Several of the more successful techniques will be discussed in Chapter 2.

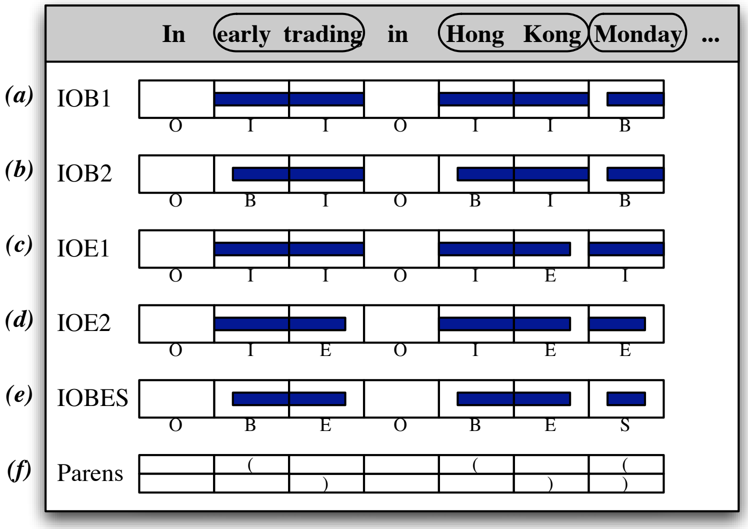


Figure 1.2: **Example Output Encodings: NP Chunking**. This figure shows six different encodings for the task of finding all "noun phrase chunks" (circled) in a sentence. Encodings *(a)*-*(e)* assign a tag to each word, indicating how that word is chunked; the meaning of each tag is indicated by the shaded bars. Encoding *(f)* uses parentheses to mark chunks. Note that a single-word chunk gets tagged with both an open and a close parenthesis tag. See Section 2.2.2 for a more detailed explanation of these encodings.

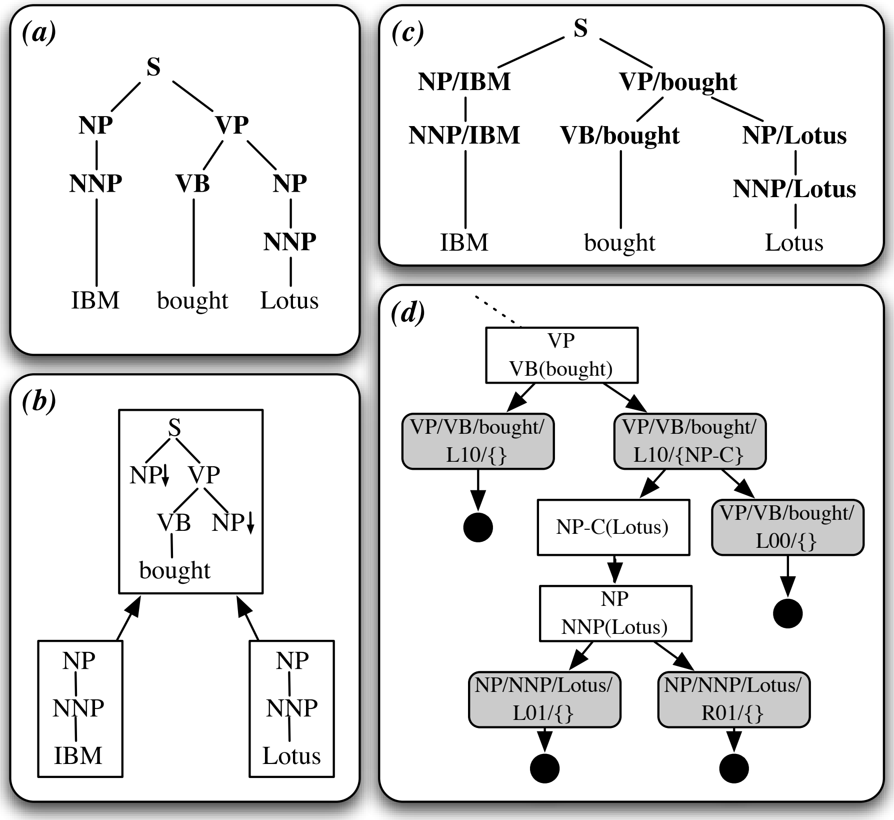


Figure 1.3: **Example Output Encodings: Parsing**. This figure shows four different encodings for the task of parsing a sentence. Structure *(a)* encodes the parse with a Treebank-style tree. Structure *(b)* encodes the parse with a TAG derivation tree (Doran et al., 1994). Structure *(c)* encodes the parse with a lexicalized tree. Structure *(d)* encodes the parse in a structure reflecting the decomposition of Michael Collin's "Model 2" parser (see Figure 2.13 for a more detailed explanation of *(d)*).

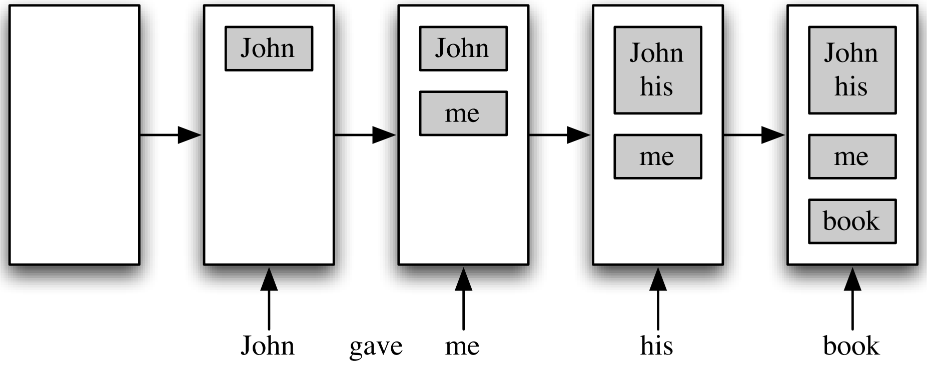


Figure 1.4: **Example Output Encoding: Coreference Resolution**. This figure shows a simplified example of an encoding for a coreference system, based loosely on Thomas Morton's coreference system (Morton, 2004). This system maintains a "discourse model" (white box), consisting of a set of "entities" (shaded boxes). Each of these entities contains a set of noun phrases, along with shared features (not shown) such as number, gender, and semantic type. Noun phrases are processed one-at-a-time, from left to right; and for each noun phrase, the system decides whether to add the noun phrase to an existing entity, or to create a new entity for it.

These techniques will form a basis for our exploration of how different problem decompositions affect the learnability of the overall problem. But the primary focus of this dissertation is on the effects of different problem decompositions, not on the learning methods used for decomposed problems.

1.1.1 Output Encodings as Transformations

Often, there is a *canonical encoding* associated with a given task or corpus, which is used to encode both the training and test data for that task or corpus. Using this canonical encoding as a reference point, we can define new encodings using reversible transformations from canonical encoded structures to a new set of encoded structures. Any reversible transformation defines a valid encoding as long as it is one-to-one – i.e., each canonical structure must correspond to exactly one transformed structure; and each transformed structure must correspond to exactly one canonical structure. We will make use of this notion of *output encoding as transformation* to define representations for specific classes of output encodings. For example, in Chapter 4, we will use finite state transducers to represent encodings of chunk structures that are based on tag sequences: a transducer defines a tag-based encoding by specifying the transformed tag sequence corresponding to each canonical tag sequence.

1.2 The Effects of Transforming Encodings

Transforming the encoding that is used to represent the output values of a task, and by extension transforming the decomposition of the task into sub-tasks, affects the accuracy with which machine learning methods can model the task and predict the correct output values for new inputs. Using the notion of "output encoding," this dissertation will examine these effects of problem decomposition on learnability, and show how they can be used to improve system performance by transforming the problem to a new encoding. These effects can be divided into two general classes: local effects and global effects. Local effects, which influence the difficulty of learning individual sub-problems, depend primarily on the coherence of the classes defined by individual output tags. Global effects, which determine the model's ability to learn long-distance dependencies, depend on the information content of the output tags.

1.3 Summary of Original Contributions

1. This dissertation will show that output encoding transformations can be used to improve performance for five structured prediction systems.

2. A novel hill-climbing algorithm is presented that can be used to automatically search for problem decompositions that improve performance, using finite state transducers as a concrete representation for output encoding transformations.

3. A novel Semantic Role Labeling system is described, which uses a single structured prediction model to jointly predict all of a verb's arguments. Output

encoding transformations are then used to extend the type of dependencies between arguments that they system can learn.

4. This dissertation will demonstrate that output encoding transformations can affect the performance of a machine learning system in two ways: by making local sub-problems more coherent; and by modifying the set of dependencies between different sub-problems that the model can learn.

5. Finally, a set of algorithms will be presented for combining sequence prediction models that use different problem decompositions, overcoming the problem that there may be no easy way to "align" the two problem decompositions. Algorithms are presented for both linear and log-linear voting.

1.4 Structure of this Document

Chapter 2 provides the background for this dissertation, including explanations of common techniques for decomposing a problem into sub-problems; and describes prior work on the effect of different output encodings, and transformations of output encodings, on the performance of supervised learning tasks. Chapter 3 shows how transforming the output space used to label semantic role labels to a more coherent output space can significantly improve performance and reduce domain specificity. Chapter 4 introduces a representation for output encodings in sequence prediction problems; and describes several experiments that use a hill-climbing algorithm to explore the space of possible encodings for three separate structured prediction tasks. Chapter 5 describes how the selection of appropriate output encodings can be used to allow a machine learning system to learn constraints and long-distance dependencies between different output elements in the task of semantic role labelling. Chapter 6 describes the issues that arise when we use voting to combine models that use different output encodings; and describes algorithms that can be used to overcome those issues. Chapter 7 summarizes the research claims of this dissertation.

Chapter 2

Background and Prior Work

In this dissertation, I will focus on three common types of task which make use of structured outputs. However, many of the results and approaches I discuss could be generalized to other related tasks. The task types I will consider are:

- *Sequence Classification*: assign a label to each input value in a given sequence.
- *Chunking*: find a set of non-overlapping sub-sequences in a given sequence.
- *Semantic Role Labelling*: identify the semantic arguments of a predicate, and label each argument's semantic role.

I will also restrict the scope of my dissertation to the class of machine learning methods which use dynamic programming to find a globally optimal output value by combining local sub-problems, where the interactions between sub-problems are mediated by output values. This class of machine learning methods includes Hidden Markov Models (HMMs); Maximum Entropy Markov Models (MEMMs); Conditional Random Fields (CRFs); and Probabilistic Chart Parsing. These machine learning methods are described in Section 2.3.

2.1 Decomposing Structured Problems

2.1.1 Bayesian Networks

The idea of probabilistically modeling a complex structured problem by breaking it into simpler sub-problems with well-defined interdependencies originates in large part from work on Belief Networks and Bayesian Networks (Pearl, 1988; Russell and Norvig, 1995; Smyth, 1998). These approaches begin by describing a given structured problem using a discrete set of variables, including measured values (inputs), latent variables, and hypothesis variables (outputs). They then use an acyclic directed graph to encode the probabilistic dependencies between those variables. Having defined this graph, they can then use it to answer probabilistic questions about the structured problem.

As an example, consider the task of modelling the following structured problem, originally described by (Pearl, 1988).

> *A person lives in a house with a burglar alarm, but is currently at work. Her burglar alarm can be set off by two possible triggers: a burglary attempt or an earthquake. When the alarm does goes off, her two neighbors, John and Mary, are each fairly reliable about calling her at work.*

First, we must describe the structured problem using a set of variables. A natural choice is the following 5 binary-valued variables: A indicates whether the alarm has gone off; E and B indicate whether there was an earthquake or burglary attempt respectively; and J and M indicate whether John or Mary respectively have called. Note that this is not the only possible decomposition of the problem into variables; for example, it would be possible to replace the variable A by two variables $A_{burglary}$ and $A_{earthquake}$, corresponding to the events of a burglary setting off the alarm and an earthquake setting off the alarm respectively.[1]

[1] In fact, it is even possible to use more "unnatural" variable decompositions, such as the following: V_1 is true iff the alarm goes off or if Mary calls; V_2 is true iff Mary calls or there is a burglary;

Having decomposed the structured problem into a set of variables, the next step is to define a graph representing the probabilistic dependencies between those variables. In order to construct this graph, we first define an ordering over the variables. In our example, this ordering is primarily motivated by the existence of *causality* links between variables. In particular, if a variable x can cause a variable y, then x should precede y in the ordering. Given this heuristic, we choose the following ordering: $\langle B < E < A < J < M \rangle$, respecting the facts that burglaries (B) and earthquakes (E) can cause the alarm to go off (A), which can in turn cause John or Mary to call (J or M). Using this variable ordering, we can decompose the joint probability distribution $P(A, B, E, J, M)$ using the chain rule:

$$P(A, B, E, J, M) = P(B)P(E|B)P(A|E, B)P(J|A, E, B)P(M|A, E, B, J) \quad (2.1)$$

We can then simplify this distribution by making several independence assumptions, again based on the notion of causality:

$$\begin{aligned} P(E|B) &= P(E) & (2.2) \\ P(J|A, E, B) &= P(J|A) & (2.3) \\ P(M|A, E, B, J) &= P(M|A) & (2.4) \end{aligned}$$

Applying these independence assumptions to our joint distribution from Equation 2.1 yields:

$$P(A, B, E, J, M) = P(B)P(E)P(A|E, B)P(J|A)P(M|A) \quad (2.5)$$

Finally, we can represent this decomposition as a graph, with a node for each variable, and with an edge $x \rightarrow y$ iff the probability for variable y is conditioned on variable x.

V_3 is true iff there is a burglary and the alarm goes off; V_4 is true iff Mary calls and there is an alarm; V_5 is true iff there is a burglary or an alarm; V_6 is true if John calls; and V_7 is true if there is an earthquake. However, such "unnatural" decompositions will severely hinder our efforts to find independencies between variables.

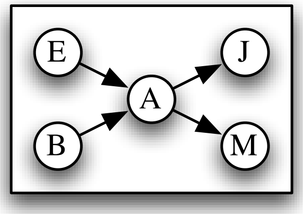


Figure 2.1: **Bayesian Network For the Alarm Problem**. Nodes in this graph represent variables: A indicates whether the alarm has gone off; E and B indicate whether there was an earthquake or burglary attempt respectively; and J and M indicate whether John or Mary respectively have called. Edges represent conditional dependencies.

2.1.2 Decomposing Structured Output Values

In (Collins, 1999), Collins discusses how the same problem decomposition techniques used to construct Bayesian networks can be applied to supervised structured learning problems. In particular, Collins proposes the following process for modelling a structured output problem:

1. **Decomposition**. Define a one-to-one mapping between output values and sequences of *decision variables*. These decision variables can be thought of as a sequence of instructions for building the output value.

2. **Independence Assumptions**. Define the conditional dependency relationships between decision variables. In (Collins, 1999), this is done by defining a function ϕ that groups conditioned decision sequences into equivalence classes.

Step (1) corresponds to decomposing a structured problem into a set of variables, and choosing an ordering for those variables. Step (2) corresponds to making independence assumptions between variables, and using those assumptions to simplify the joint distribution model.

The main difference between simple Bayesian networks and supervised structured learning problems is that for Bayesian networks, we are working with a fixed graph;

but for supervised structured learning problems, we define a separate (but related) graph for each possible output value. In other words, supervised structured learning can be thought of as an attempt to model output values using a *family* of Bayesian networks, and to choose the most likely Bayesian network for a given input value.

Another important difference between Bayesian networks like the example in Section 2.1.1 and supervised structured learning is that there is typically not a natural notion of *causality* that we can apply when deciding how to decompose structured values. However, Collins proposes that we can generalize the notion of causality to the notion of *locality*, where the *domain of locality* of an entity is the set of entities that it can directly effect. Thus, when deciding how to decompose a structured output value, we should attempt to maintain structural connections between any variables that are within each others' domain of locality. In the case of parsing, Collins uses this assumption to justify a decomposition based on head-word based dependencies and subcategorization frames.

2.2 Structured Output Tasks

2.2.1 Sequence Classification

A sequence classification task is any task that consists of mapping a sequence of input values to a corresponding sequence of labels. Typically, the label sequence will be the same length as the input sequence. Examples of sequence classification tasks include part-of-speech tagging, which labels each word in a sentence with a tag describing its lexical category; and activity classification, which classifies the type of activity an agent is performing during different subsets of a time sequence.

It should be noted that performing sequence classification is not equivalent to performing a sequence of simple classification tasks, because there are dependencies between the different label values. Thus, a sequence of labels that is formed by taking labels that are each likely individually may not be a likely sequence overall if it violates constraints between the labels.

2.2.2 Chunking

A chunking task is any task that consists of finding some set of non-overlapping subsequences in a given sequence. Examples of chunking tasks include named entity detection, which searches a text for proper nouns; noun phrase chunking, which identifies non-recursive noun phrase chunks in a sentence; and gene intron detection, which searches DNA for gene sequences that encode for proteins.

Chunking tasks are typically used as an initial step in a natural language processing system, to find entities of interest which can then be examined further. For example, information extraction systems often use chunking subsystems to find mentions of the people and places in a document; after these mentions have been located, the system can then attempt to determine how they relate to one another.

The most common encodings for chunking tasks associate a single tag with each input token. The most popular chunking encodings for machine learning tasks are `IOB1` and `IOB2`, both of which make use of the following three tags:

- `I`: This token is *inside* (i.e., part of) a chunk.
- `O`: This token is *outside* (i.e., not part of) a chunk.
- `B`: This token is at the *beginning* of a chunk.

The difference between `IOB1` and `IOB2` is that `IOB2` uses the `B` tag at the beginning of all chunks, while `IOB1` only uses the `B` tag at the beginning of chunks that immediately follow other chunks.[2] The tag sequences generated by these encodings for a sample sentence are shown in the first two lines of Figure 2.2.

It should be noted that the set of valid tag sequences for each of these two encodings does *not* include all sequences of `I`, `O`, and `B`. In particular, the `IOB1`

[2] Note that an encoding that just used the `I` and `O` tags would be incapable of distinguishing two adjacent one-element chunks from a single two-element chunk.

	In	**early**	**trading**	**in**	**Hong**	**Kong**	**Monday**	**...**
IOB1	O	I	I	O	I	I	B	...
IOB2	O	B	I	O	B	I	B	...
IOE1	O	I	I	O	I	E	I	...
IOE2	O	I	E	O	I	E	E	...
IOBES	O	B	E	O	B	E	S	...

Figure 2.2: **Common Chunking Encodings**. Five common chunking encodings for an example sentence, drawn from the Ramshaw & Marcus noun phrase chunking corpus (Ramshaw and Marcus, 1995). See Figure 1.2 for a graphical depiction of this example.

encoding will never generate a tag sequence including the sub-sequence OB; and IOB2 encoding will never generate a tag sequence including the sub-sequence OI. However, it is common practice to allow machine learning systems to generate these technically invalid tag sequences, and to simply "correct" them. In particular, when using IOB1, the tag sequence OB is corrected to OI; and when using IOB2, the tag sequence OI is corrected to OB. This is typically the right thing to do, since machine learning algorithms are usually more likely to confuse I and B than to confuse O with I or B.

An alternative chunking encoding that is sometimes used is to mark the chunks' end tokens instead of their beginning tokens. The IOE1 and IOE2 encodings use the E tag to mark the final token of chunks. In IOE2, the final token of every chunk is marked, while in IOE1, the E tag is only used for chunks that immediately precede other chunks. An example of the tag sequences generated by these two encodings is shown in Figure 2.2.

Several other chunking encodings have also been proposed. One common variant is to mark both the beginning and the end of all chunks. Since a single-token chunk is both the beginning and the end of a chunk, it is given a new tag, S (for "singleton"). I will refer to this five-tag encoding as IOBES.

In	**early**	**trading**	**in**	**Hong**	**Kong**	**Monday**	**...**
O	O	O	O	I-place	I-place	B-date	...

Figure 2.3: **Example Labeled Chunk Encoding**. This encoding is formed by first identifying each chunk using the IOB1 tags; and then appending a class label (such as `person` or `place`) to each tag in a chunk, identifying that chunk's type.

Labeled Chunking

For many problems, we are interested in associating a single label with each chunk. A "labeled chunking task" is any task that consists of finding some set of non-overlapping sub-sequences in a given sequence, and assigning a single label to each such sub-sequence. A common example of a labeled chunking task is labeled named entity detection, which involves finding all proper names in a document, and labeling each one with a label indicating its type, such as "person," "place," or "organization."

Output structures for labeled chunking tasks are typically encoded using a variant of one of the basic chunking encodings, where each basic tag (such as `B`, `I`, or `O`) is combined with a label (such as `person`). All of the tags corresponding to a given chunk are required to use the same label. Figure 2.3 shows an example of a labeled chunk encoding for the named entity detection task.

2.2.3 Semantic Role Labelling

Correctly identifying semantic entities and successfully disambiguating the relations between them and their predicates is an important and necessary step for successful natural language processing applications, such as text summarization, question answering, and machine translation. For example, in order to determine that question (1a) is answered by sentence (1b), but not by sentence (1c), we must determine the relationships between the relevant verbs (*eat* and *feed*) and their arguments.

(1) a. What do lobsters like to eat?

b. Recent studies have shown that lobsters primarily feed on live fish, dig for clams, sea urchins, and feed on algae and eel-grass.

c. In the early 20th century, Mainers would only eat lobsters because the fish they caught was too valuable to eat themselves.

An important part of this task is *Semantic Role Labeling* (SRL), where the goal is to locate the constituents which are arguments of a given verb, and to assign them appropriate semantic roles that describe how they relate to the verb.

PropBank

PropBank (Palmer et al., 2005) is an annotation of one million words of the Wall Street Journal portion of the Penn Treebank II (Marcus et al., 1994) with predicate-argument structures for verbs, using semantic role labels for each verb argument. In order to remain theory neutral, and to increase annotation speed, role labels were defined on a per-lexeme basis. Although the same tags were used for all verbs, (namely Arg0, Arg1, ..., Arg5), these tags are meant to have a verb-specific meaning.

Thus, the use of a given argument label should be consistent across different uses of that verb, including syntactic alternations. For example, the Arg1 (underlined) in "John broke the window" has the same relationship to the verb as the Arg1 in "The window broke", even though it is the syntactic subject in one sentence and the syntactic object in the other.

But there is no guarantee that an argument label will be used consistently across different verbs. For example, the Arg2 label is used to designate the *destination* of the verb "bring;" but the *extent* of the verb "rise." Generally, the arguments are simply listed in the order of their prominence for each verb. However, an explicit effort was made when PropBank was created to use Arg0 for arguments that fulfill Dowty's criteria for "prototypical agent," and Arg1 for arguments that fulfill the criteria for "prototypical patient" (Dowty, 1991). As a result, these two argument labels are significantly more consistent across verbs than the other three. But nevertheless, there are still some inter-verb inconsistencies for even Arg0 and Arg1.

PropBank divides words into lexemes using a very coarse-grained sense disambiguation scheme: two senses are only considered different if their argument labels are different. For example, PropBank distinguishes the "render inoperable" sense of "break" from the "cause to fragment" sense. In PropBank, each word sense is known as a "frame." Information about each frame, including descriptions of the verb-specific meaning for each argument tag (Arg0, . . . , Arg5), is defined in "frame files" that are distributed with the corpus.

The primary goal of PropBank is to provide consistent general purpose labeling of semantic roles for a large quantity of coherent text that can provide training data for supervised machine learning algorithms, in the same way the Penn Treebank has supported the training of statistical syntactic parsers. PropBank can provide frequency counts for (statistical) analysis or generation components for natural language applications. In addition to the annotated corpus, PropBank provides a lexicon which lists, in the frame file for each annotated verb, for each broad meaning, its "frameset", i.e., the possible arguments in the predicate and their labels and possible syntactic realizations. This lexical resource is used as a set of verb-specific guidelines by the annotators, and can be seen as quite similar in nature to FrameNet, although much more coarse-grained and general purpose in the specifics.

PropBank's Relationship to Dependency Parsing

PropBank's model of predicate argument structures differs from dependency parsing in that it is applied on a per-verb basis: in dependency parsing, each phrase can be dependent on only one other phrase; but since PropBank describes each verb in the sentence independently, a single argument may be used for multiple predicates. For example, in the following sentence, PropBank would use the phrase "his dog" as the argument to two predicates, "scouted" and "chasing:"

(2) a. His dog **scouted** ahead, chasing its own mangy shadow.

b. His dog scouted ahead, **chasing** its own mangy shadow.

Output Tags	$Y = Y_1, Y_2, ..., Y_n$
Input Feature Vectors	$X = X_1, X_2, ..., X_m$
Input Sequence	$\vec{x} = x_1, x_2, ..., x_T, x_i \in X$
Output Sequence	$\vec{y} = y_1, y_2, ..., y_T, y_i \in Y$

Figure 2.4: **Notation for Sequence Learning**.

2.3 Sequence Learning Models

Sequence learning models are designed to learn tasks where each output is decomposed into a linear sequence of tags. For example, these models can be applied to chunking tasks that have been encoded using `IOB1` or `IOB2`. Sequence learning models take a sequence of input values, and must predict the most likely sequence of output tags for that input sequence. In particular, each *task instance* is of a pair $(\vec{x}, \vec{y})$, where $\vec{x} = x_1, x_2, ..., x_T$ is a sequence of feature vectors describing the input value; and $\vec{y} = y_1, y_2, ..., y_T$ is a sequence of output tags, encoding the structured output value $\mathbf{y} = \text{encode}(\vec{y})$.[3] Models are trained using a corpus of task instances, by maximizing the likelihood of the instance outputs given their inputs. Models can then be tested using a separate corpus of task instances, by running them on the instance inputs, and comparing the model's outputs to the instance outputs. Evaluation metrics for comparing these two output values are discussed in Section 2.3.4.

In this dissertation, I will make use of three sequence learning models: Hidden Markov Models (HMMs); Maximum Entropy Markov Models (MEMMs); and Linear Chain Conditional Random Fields (CRFs). These three models share several characteristics:

1. They are all probabilistic models.

2. They all rely on the Markov assumption, which states that the probability of a sequence element given all previous elements can be approximated as

[3] In general, the length of the input sequence is not required to be equal to the length of the output sequence; but for the purposes of this dissertation, I will restrict my attention to sequence learning tasks where $\text{len}(\vec{x}) = \text{len}(\vec{y})$.

the probability of that sequence element given just the immediately preceding element.[4]

3. They all use dynamic programming to find the most likely output sequence for a given input.

2.3.1 Hidden Markov Models

A Hidden Markov Model (HMM) is a sequence learning model predicated on the assumption that task instances are generated by a discrete Markov process. A *discrete Markov process* is a graphical process with a set of N distinct states $s_1, s_2, ..., s_N$ and M distinct symbols $k_1, k_2, ..., k_M$. Over time, this process transitions through a sequence of states, and simultaneously generates a corresponding sequence of symbols. HMMs model sequence learning tasks as discrete Markov processes, where states are used to represent output tags, and symbols are used to represent input feature vectors. Thus, the probability assigned to a given task instance $(\vec{x}, \vec{y})$ is equal to the probability that the Markov process transitions through the state sequence $\vec{y}$ while generating the symbol sequence $\vec{x}$.

The transition and generation probabilities of a discrete Markov process are fixed, and do not vary with time. At time $t = 1$, the process starts in state $y_1 \in S$ with probability π_{y_1}. At each time step t, the process transitions from its current state y_t to state y_{t+1} with probability $a_{y_t y_{t+1}}$. Thus, the probability that the Markov process generates any given state sequence $\vec{y} = (y_1, ..., y_T)$ is given by:

$$P(\vec{y}) = \pi_{y_1} \prod_{t=1}^{T-1} a_{y_t y_{t+1}} \tag{2.6}$$

As the Markov process transitions through a sequence of states, it generates a corresponding sequence of symbols. At each time t, the process generates a single

[4] Or more generally, that the probability of an element given all previous elements can be approximated as the probability of that sequence element given just the immediately preceding n elements, for some fixed value of n.

Symbol alphabet	$K = \{k_1, ..., k_M\}$	
Set of states	$S = \{s_1, ..., s_N\}$	
Generated symbol sequence	$\vec{x} = (x_1, ..., x_T)$	$x_t \in K, t \in \{1, 2, ..., T\}$
State sequence	$\vec{y} = (y_1, ..., y_T)$	$y_t \in S, t \in \{1, 2, ..., T\}$
Output value	$\mathbf{y} = \text{encode}(\vec{y})$	
Initial state probabilities	$\Pi = \{\pi_s\}$	$s \in S$
State transition probabilities	$A = \{a_{s_i s_j}\}$	$s_i \in S, s_j \in S$
Symbol emission probabilities	$B = \{b_s(k)\}$	$s \in S, k \in K$
Training corpus	$\langle X, Y \rangle$	$X = \left(\vec{x}^{(1)}, \vec{x}^{(2)}, ..., \vec{x}^{(N)}\right)$ $Y = \left(\vec{y}^{(1)}, \vec{y}^{(2)}, ..., \vec{y}^{(N)}\right)$

Figure 2.5: **Notation for Hidden Markov Models**.

symbol $x_t \in K$ with probability $b_{y_t}(x_t)$. Thus, the probability that the Markov process generates a given task instance $(\vec{x}, \vec{y})$ is:

$$P\left(\vec{y}, \vec{x}\right) = \pi_{y_1} \prod_{t=1}^{T-1} a_{y_t y_{t+1}} \prod_{t=1}^{T} b_{y_t}(x_t) \tag{2.7}$$

HMM Training

HMMs are trained by setting the three probability distributions Π, A, and B based on a training corpus $\langle X, Y \rangle$. The initial state probabilities Π are initialized by simply counting how many of the training instances begin with each state s, and dividing by the total number of training instances:

$$\begin{aligned} \pi_s &= \widehat{P}(y_1 = s) & (2.8) \\ &= \frac{count(y_1 = s)}{N} & (2.9) \end{aligned}$$

Similarly, the state transition probabilities A are set by counting how often the Markov process transitions from state s_i to s_j, and dividing by the total number of

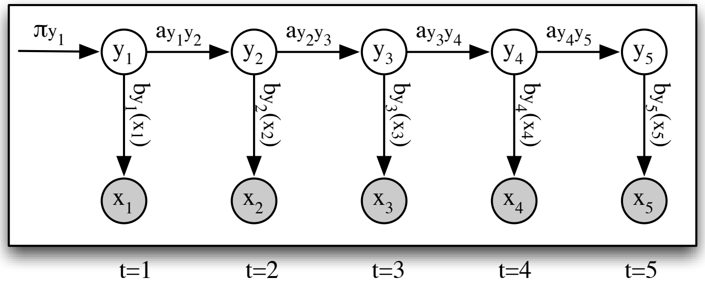


Figure 2.6: **HMM as a Bayesian Graphical Model**. This figure shows how HMMs are related to Bayesian Networks. Nodes are used to represent variables: the nodes marked y_t represent the states at each time step; and the nodes marked x_t represent the emitted symbols at each time step. Edges represent statistical dependencies between variables, and are labeled with probabilities. The length of the Bayesian Network chain will depend on the length of the individual instance; in this case, the instance has a length of 5.

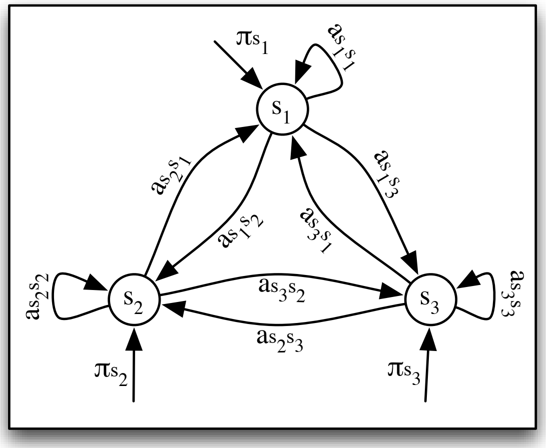


Figure 2.7: **HMM as a Finite State Machine**. This graphical depiction of an HMM highlights its relationship to finite state machines. This HMM has three states, s_1, s_2, and s_3. Arcs are labeled with probabilities: the arcs marked with π_i indicate that the HMM may start in any of the three states, with the given probabilities; and the arcs between states indicate the probability of transitioning between states. Symbol emission probabilities are not shown.

outgoing transitions from state s_i:

$$
\begin{aligned}
a_{s_i s_j} &= \widehat{P}(y_t = s_i, y_{t+1} = s_j) && (2.10)\\
&= \frac{count(y_t = s_i, y_{t+1} = s_j)}{count(y_t = s_i)} && (2.11)
\end{aligned}
$$

However, the symbol emission probabilities typically can not be modeled by simple counting: because there are usually a very large number of possible feature vector values, these counts would be too low to reliably estimate the distribution. Instead, the symbol emission probabilities are usually modeled using a generative classifier model, such as Naive Bayes.

HMM Decoding

Once an HMM has been trained, it can be used to predict output values for new inputs. In particular, the predicted output value $\vec{y}^*$ for a given input $\vec{x}$ is simply the output value that maximizes $P(\vec{y}|\vec{x})$:

$$\vec{y}^* = \arg\max_{\vec{y}} P(\vec{y}|\vec{x}) \tag{2.12}$$

$\vec{y}^*$ can be computed efficiently using a dynamic programming technique known as *Viterbi decoding*. This same technique will also be used to predict output values for MEMMs and linear chain CRFs. First, we will construct a graphical structure called a *Viterbi graph*, which combines the HMM's three probability distributions a, b, and π, into a single graph. This graph is specific to a single input value $\vec{x}$; i.e., each input value $\vec{x}$ will have its own Viterbi graph. Each node in the graph represents an assignment of a single output tag, as indicated by the node labels; and paths through the graph represent assignments of output tag sequences. The edges are annotated with weights that combine the HMM's three probability distributions, as follows:

$$
\begin{aligned}
v_s(1) &= \pi_s b_s(x_1) && (2.13)\\
v_{s_i s_j}(t) &= a_{s_i s_j} b_{s_j}(x_t) \qquad 2 \leq t \leq T && (2.14)
\end{aligned}
$$

Viterbi Graph	$\langle S, T, Q, E\rangle$		
Viterbi Graph Nodes	Q	$=$	$\{q_0\} \cup \{q_{t,s} : 1 \leq t \leq T; s \in S\}$
Viterbi Graph Edges	E	$=$	$\{\langle q_0 \rightarrow q_{1,s}\rangle : s \in S\} \cup$
			$\{\langle q_{t-1,s} \rightarrow q_{t,s'}\rangle : s \in S; t \in T\}$
Viterbi Graph Edge Weights	$v_s(1)$	$=$	weight $(q_0 \rightarrow q_{1,s})$
	$v_{ss'}(t)$	$=$	weight $(q_{t-1,s} \rightarrow q_{t,s'})$
Max Forward Scores	$\delta_s(t)$		
Max Backward Scores	$\phi_s(t)$		

Figure 2.8: **Notation for Viterbi Graphs**.

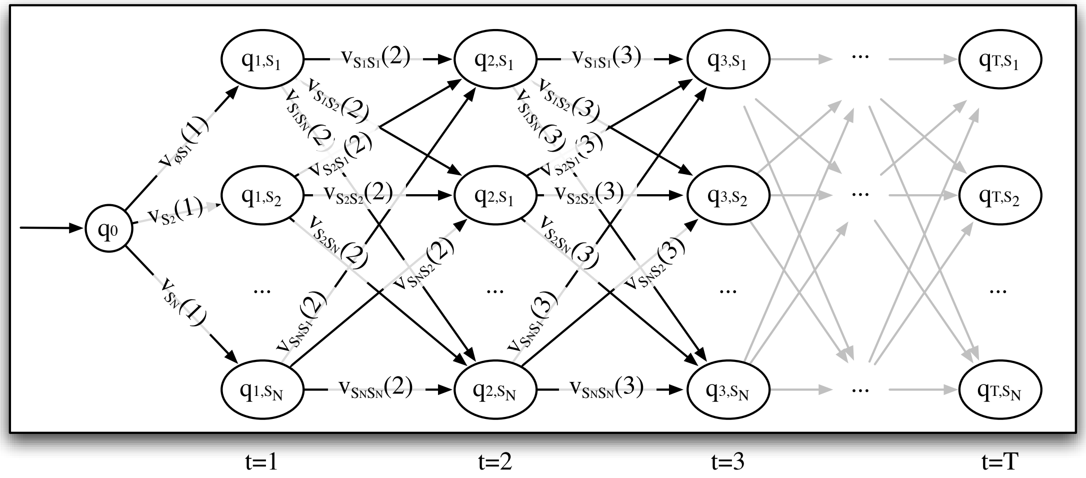


Figure 2.9: **Viterbi Graph**. This graphical structure is used for *decoding*, or finding the most likely output value, in HMMs, MEMMs, and linear chain CRFs. Each node q_{t,s_i} represents an assignment of a single output tag $y_t = s_i$. Edges are annotated with weights, such that the score of an output value is equal to the product of edge weights in the corresponding path. Using dynamic programming, we can find the output value that maximizes this score.

Using these edge weights, the probability of an output value $\vec{y}$ is simply the product of the edge weights in the corresponding path:

$$\begin{aligned} P(\vec{y}, \vec{x}) &= \pi_{y_1} \prod_{t=1}^{T-1} a_{y_t y_{t+1}} \prod_{t=1}^{T} b_{y_t}(x_t) && (2.15) \\ &= (\pi_{y_1} b_{y_1}(x_1)) \left(\prod_{t=2}^{T} a_{y_{t-1} y_t} b_{y_t}(x_t) \right) && (2.16) \\ &= v_{y_1}(1) \prod_{t=2}^{T} v_{y_{t-1} y_t}(t) && (2.17) \end{aligned}$$

In order to find the output value $\vec{y}^*$ that maximizes this probability, we use a dynamic programming algorithm based on a new variable $\delta_s(t)$, known as the *max forward score*[5]:

$$\begin{aligned} \delta_s(1) &= v_s(1) && (2.18) \\ \delta_s(t) &= max_{s'} \delta_{t-1}(s') v_{s's}(t) \qquad 1 < t \leq T && (2.19) \end{aligned}$$

This variable contains the score of the highest scoring path from the start node q_0 to the node $q_{t,s}$ (where a path score is the product of edge weights in that path). We can find the highest scoring path (and thus the most likely output value) by backtracking through the graph and maximizing over $\delta_s(t)$:

$$\begin{aligned} y_T^* &= \arg\max_s \delta_s(T) && (2.20) \\ y_t^* &= \arg\max_s \delta_t(s) v_{s \vec{y}_{t+1}^*}(t) \qquad 1 \leq t < T && (2.21) \end{aligned}$$

We will also define the max backward score $\phi_s(t)$ to contain the score of the highest scoring path from the node $q_{t,s}$ to the end of the graph.

$$\begin{aligned} \phi_s(T) &= 1 && (2.22) \\ \phi_s(t) &= max_{s'} v_{ss'}(t+1) \phi_{t+1}(s') \qquad 1 \leq t < T && (2.23) \end{aligned}$$

Thus, the score of the highest scoring path that passes through node $q_{t,s}$ is $\delta_s(t)\phi_s(t)$.

[5] I use the term *score* rather than *probability* because Viterbi graphs do not always encode probabilities (e.g., in CRFs)

2.3.2 Maximum Entropy Markov Models

Maximum Entropy Markov Models (MEMMs) are very similar in structure to HMMs. They differ in that the HMM state transition and symbol emission distributions are replaced by a Maximum Entropy (ME) model. This model is used to find the probability that the output value contains a specified state, given the previous state and the current input value:

$$P(y_t|y_{t-1}, x_t) \tag{2.24}$$

This distribution is modelled using an exponential model combining weighted features of x_t, y_t, and y_{t-1}:

$$P(y_t|y_{t-1}, x_t) = \frac{1}{Z} exp\left(\sum_{a \in A} \lambda_a f_a(x_t, y_t, y_{t-1})\right) \tag{2.25}$$

Where A is the set of feature identifiers, f_a are feature functions, λ_a are learned feature weights, and Z is a normalizing constant.

Alternatively, the probability distribution (2.24) can be modelled using a separately trained model for each value of y_{t-1}:

$$P(y_t|y_{t-1}, x_t) = P_{y_{t-1}}(y_t|x_t) = \frac{1}{Z} exp\left(\sum_{a \in A_{y_{t-1}}} \lambda_a f_a(x_t, y_t)\right) \tag{2.26}$$

A significant advantage of MEMMs over HMMs is that they do not rely on the assumption that all features are mutually independent. Additionally, features may be defined that combine information about the current input value x_t and the previous output tag y_{t-1}.

MEMM Training

MEMMs are trained by building the underlying Maximum Entropy model or models. These models can be trained using a wide variety of optimization methods, such as iterative scaling methods (GIS, IIS) and conjugate gradient methods (McCallum et al., 2000).

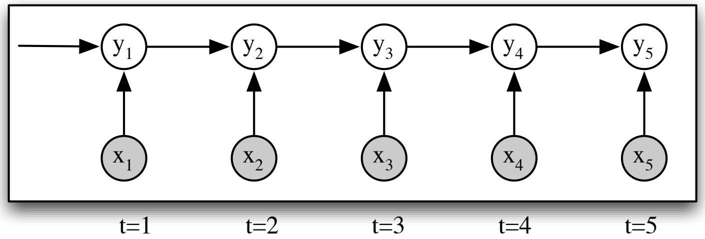


Figure 2.10: **MEMM as a Bayesian Graphical Model**. This figure shows how MEMMs are related to Bayesian Networks. Nodes are used to represent variables: the nodes marked y_t represent the states at each time step; and the nodes marked x_t represent the emitted symbols at each time step. Edges represent statistical dependencies between variables. The length of the Bayesian Network chain will depend on the length of the individual instance; in this case, the instance has a length of 5.

MEMM Decoding

MEMM decoding is very similar to HMM decoding. In particular, we can find the most likely output value $\vec{y}^*$ for a given input $\vec{x}$ by applying the Viterbi algorithm (described in Section 2.3.1) to a Viterbi graph with the following edge weights:

$$v_s(1) = \quad P(s|x_1) \quad = \frac{1}{Z} exp\left(\sum_a \lambda_a f_a(x_1, s)\right) \tag{2.27}$$

$$v_{s_i s_j}(t) = \quad P(s_j|s_i, x_t) \quad = \frac{1}{Z} exp\left(\sum_a \lambda_a f_a(x_t, s_j, s_i)\right) \qquad 2 \leq t \leq T \tag{2.28}$$

MEMMs (and linear chain CRFs, described in the next section) differ from HMMs in two important ways:

- MEMMs (and linear chain CRFs) model the conditional distribution $P(\vec{y}|\vec{x})$ directly, rather than deriving this conditional distribution from a model of the generative distribution $P(\vec{y}, \vec{x})$. As a result, the model has fewer free parameters, which may make it less susceptible to over-fitting.
- Because MEMMs (and linear chain CRFs) are conditional models, their features may depend on the entire input value $\vec{x}$, rather than just the local input

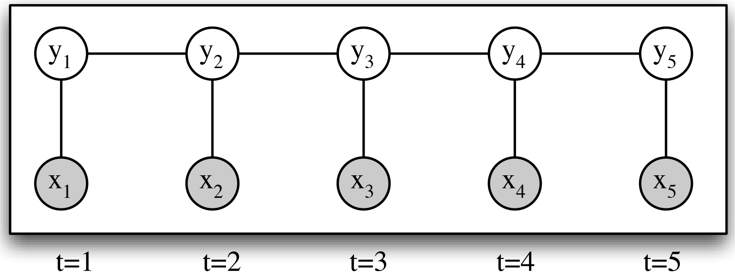


Figure 2.11: **Linear Chain CRF as a Bayesian Graphical Model**. This figure shows how Linear Chain CRFs are related to Bayesian Networks. Nodes are used to represent variables: the nodes marked y_t represent the states at each time step; and the nodes marked x_t represent the emitted symbols at each time step. Edges represent statistical dependencies between variables. The length of the Bayesian Network chain will depend on the length of the individual instance; in this case, the instance has a length of 5.

value x_t.

2.3.3 Linear Chain Conditional Random Fields

Linear chain Conditional Random Fields (CRFs) are similar to both HMMs and MEMMs in their basic structure. The main difference between linear chain CRFs and MEMMs is that linear chain CRFs use a single globally normalized model for the entire input, rather than using a locally normalized models for each point in the Viterbi graph. This helps to prevent the "label bias problem," which can cause MEMMs to give a high score to a state transition even if the model knows that the transition is quite unlikely.

The conditional probability distribution defined by a linear chain CRF is:

$$P\left(\vec{y}|\vec{x}\right) = \frac{1}{Z(x)} exp\left(\sum_{t=1}^{T}\sum_{a\in A}\lambda_a f_a(\vec{x}, y_t, y_{t-1}, t)\right) \tag{2.29}$$

Where A is the set of feature identifiers, f_a are feature functions, λ_a are learned feature weights, and $Z(x)$ is an input-specific normalizing constant.

Linear Chain CRF Training

Linear Chain CRFs are trained using a wide variety of optimization methods, such as iterative scaling methods (GIS, IIS) and conjugate gradient methods (Sutton and McCallum, 2006). These methods all attempt to find the set of weights that maximize the log-likelihood of a given training corpus $(\vec{x}_k, \vec{y})_{k}{}_{k=1}^{N}$:

$$\begin{aligned} \lambda* &= \arg\min_{\lambda} \left(\sum_{k} p_{\lambda}(\vec{y}_k | \vec{x}_k) \right) && (2.30) \\ &= \arg\min_{\lambda} \left(\sum_{k} [\lambda \cdot F(\vec{y}_k, \vec{x}_k) - logZ_{\lambda}(\vec{x}_k)] \right) && (2.31) \end{aligned}$$

Linear Chain CRF Decoding

As with HMMs and MEMMs, decoding is performed by constructing a Viterbi graph capturing the likelihood scores for a given input, and using the Viterbi algorithm to find the most likely output. For Linear chain CRFs, we set the Viterbi graph edge weights as follows:

$$\begin{aligned} v_s(1) &= exp\left(\sum_{a \in A} \lambda_a f_a(\vec{x}, y_1, 1) \right) && (2.32) \\ v_{s_i s_j}(t) &= exp\left(\sum_{a \in A} \lambda_a f_a(\vec{x}, y_t, y_{t-1}, t) \right) \qquad 2 \leq t \leq T && (2.33) \end{aligned}$$

Two things are worth noting about this Viterbi graph definition. First, unlike the Viterbi graphs for HMMs and CRFs, individual edges in the graph do not correspond to any probabilistic value; it is only when we combine a complete path through the graph that we arrive at a meaningful score. Second, the normalization factor $Z(\vec{x})$ is not included in the Viterbi graph. Thus, if we want to find the predicted probability of a particular output value, we would need to adjust the path's score by dividing by $Z(\vec{x})$:

$$P(\vec{y}|\vec{x}) = \frac{1}{Z(x)} v_{y_1}(1) \prod_{t=2}^{T} v_{y_{t-1}y_t}(t) \tag{2.34}$$

But since we are generally only interested in determining the highest scoring output value $\vec{y}^*$, and since $Z(\vec{x})$ is constant across all values of $\vec{y}$ for a given $\vec{x}$, we typically don't need to compute $Z(\vec{x})$:

$$\begin{aligned}
\vec{y}^* &= \operatorname*{arg\,max}_{vecy} P(\vec{y}|\vec{x}) && (2.35)\\
&= \operatorname*{arg\,max}_{vecy} \frac{1}{Z(x)} v_{y_1}(1) \prod_{t=2}^{T} v_{y_{t-1}y_t}(t) && (2.36)\\
&= \operatorname*{arg\,max}_{vecy} v_{y_1}(1) \prod_{t=2}^{T} v_{y_{t-1}y_t}(t) && (2.37)
\end{aligned}$$

2.3.4 Evaluating Sequence Models

A number of different metrics can be used to evaluate the performance of a sequence modelling system. All of these metrics assume the existence of a *test corpus* $\langle X, Y \rangle$, where $X = \left(\vec{x}^{(1)}, \vec{x}^{(2)}, ..., \vec{x}^{(N)}\right)$ is a list of input values, and $Y = \left(\vec{y}^{(1)}, \vec{y}^{(2)}, ..., \vec{y}^{(N)}\right)$ is a list of the corresponding output values. i.e., the correct output for $\vec{x}^{(i)}$ is $\vec{y}^{(i)}$. In order to evaluate a given system, we will use that system to predict the most likely output value $\widehat{\vec{y}}^{(i)}$ for each input $\vec{x}^{(i)}$; and then compare those predicted output values to the correct output values.

The simplest metric computes the accuracy over corpus instances:

$$acc_{instance}(\langle X, Y \rangle, \widehat{Y}) = \frac{\text{count}\left(\vec{y}^{(i)} = \widehat{\vec{y}}^{(i)}\right)}{N} \tag{2.38}$$

However, this metric is not often used, because it does not give any partial credit to "mostly correct" solutions. In particular, all incorrect outputs are treated the same, whether they differ from the correct output in one tag or in all tags. Therefore, a more common metric is to evaluate the accuracy over tags in the corpus:

$$acc_{tag}(\langle X, Y\rangle, \widehat{Y}) = \frac{\text{count}\left(y_t^{(i)} = \widehat{y}_t^{(i)}\right)}{\text{count}\left(y_t^{(i)}\right)} \tag{2.39}$$

But one disadvantage of evaluating systems based on individual tags is that it removes some of the incentive to find outputs that are globally plausible. For example, optimizing a part-of-speech tagger for acc_{tag} may result in a sequence of part-of-speech tags that are plausible when examined individually, but highly unlikely when taken as a whole.

A middle-ground between $acc_{instance}$ and acc_{tag} is possible for tasks where a system's output can be thought of as a set of elements. For example, the chunking task can be thought of as producing a set of chunks, each of which is uniquely defined by a span of words in the sentence. In such tasks, we can evaluate systems by comparing the set of elements generated by the system, $elements(\widehat{\vec{y}}^{(i)})$, to the correct set of elements for that input, $elements(\vec{y}^{(i)})$.

$$precision = \frac{\text{count}(elements(\vec{y}^{(i)}) \cap elements(\widehat{\vec{y}}^{(i)}))}{\text{count}(elements(\widehat{\vec{y}}^{(i)}))} \tag{2.40}$$

$$recall = \frac{\text{count}(elements(\vec{y}^{(i)}) \cap elements(\widehat{\vec{y}}^{(i)}))}{\text{count}(elements(\vec{y}^{(i)}))} \tag{2.41}$$

Precision evaluates how many of the predicted elements are correct elements; and recall evaluates how many of the correct elements were generated. A final metric, F_α, combines these two scores by taking their weighted harmonic mean:

$$F_\alpha = \frac{(1+\alpha) \cdot precision \cdot recall}{\alpha \cdot precision + recall} \tag{2.42}$$

$$\tag{2.43}$$

2.4 Higher Order Sequence Learning Models

In a traditional (or "first order") HMM, the probability of transitioning to a state depends only on the identity of the immediately previous state. As a result, first

Label sequence	A	B	A	C	C
State sequence (2^{nd} order)	SA	AB	BA	AC	CC
State sequence (3^{rd} order)	SSA	SAB	ABA	BAC	ACC

Figure 2.12: **Implementing Higher Order HMMs as an Output Encoding Transform**. Each state consists of a series of n consecutive labels, recording the history of labels that has been predicted within a fixed window. The special symbol S (start) is used when the history window extends past the beginning of the sentence.

order HMMs are unable to model systems whose transition probabilities depend on states prior to the immediately previous state. However, it is possible to get around this problem, and allow the transition probabilities to depend on states within a fixed history window, by making use of a second order, third order, or any n^{th} order HMM.

2.4.1 Implementing Higher Order Models Using Output Transformations

In these "higher order" HMMs, the probability of transitioning to a state depends on the identity of the n previous states. Although it is possible to model these dependencies directly, it is more common to implement higher order HMMs using an output transformation.

In particular, rather than defining a state corresponding to each individual label, we can define states that correspond to sequences of n labels. Each of these states records the history of labels that has been predicted within a fixed history window of size n. For example, Figure 2.12 shows how a sequence of labels would be transformed to a corresponding sequence of states. In this transformed model, the probability of transitioning from one state to the next depends only on the identity of the immediately previous state. But since the immediately previous state contains the identity of the n previous labels, we can encode transition probabilities that depend on the n previous labels.

This technique of generating a higher-order model by transforming the label sequence can also be applied to other sequence prediction models, such as MEMMs and Linear Chain CRFs. As was the case with HMMs, this transformation to a higher-order model will allow these learning algorithms to model systems whose transition probabilities depend on labels prior to the immediately previous label.

2.4.2 Other Transformations

As we have seen, simple transformations on the output encoding can be used to replace a first-order model with a higher-order model that can capture longer-distance dependencies between output labels. In the remainder of this dissertation, we explore the effects that a wide variety of other transformations can have on the "model structure," and on the ability of learning algorithms to accurately model interesting problems.

2.5 Prior Work: Encoding Classification Output Values

In *classification tasks*, a model must learn to label each input value with a single tag, drawn from a fixed tag set. Thus, the set of possible output values is relatively small, when compared with structured output tasks. There have been a number of attempts to improve performance of classification models by transforming the representation of these output tags.

2.5.1 Error Correcting Output Codes

One such attempt is Error Correcting Output Codes (Dietterich and Bakiri, 1995), which decomposes a single classification task into a set of subtasks that are implemented by base learners . Each of these base learners is trained to distinguish different subsets of output values. The output of these individual base learners is then combined in such a way that the correct output tag will be generated even if one or two of the base learners makes an incorrect prediction.

In particular, if we encode the outputs of the individual learners as bit strings, indicating which value each individual learner picked, then we can assign a class to a new value by choosing the class whose bit string most closely matches the output for that new value. In order to maximize the robustness of this system, Dietterich & Bakiri use problem decompositions that maximize the minimum Hamming distance between any two class's bit strings. In other words, the classifiers are defined in a way that maximizes the number of classifiers that would need to generate incorrect outputs before the overall output would be incorrect. For example, (Dietterich and Bakiri, 1995) define a system for identifying hand-written numbers (0-9) using 15 subproblems, each of which distinguishes a different subset of the digits. By maximizing the Hamming distance between the class's bit strings, Dietterich & Bakiri ensure that at least 3 separate classifiers would need to generate incorrect outputs for the overall system to assign the wrong class.

2.5.2 Multi-Class SVMs

Another line of work that has examined different ways to decompose a multi-way classification into subproblems comes from work on binary classifiers. For example, by their nature SVMs are restricted to making binary classification decisions. In order to build a multi-way classifier with SVMs, the multi-way classification problem must first be decomposed into a set of binary classification decision subproblems. SVM models can then be trained for each of these subproblems; and the results combined to generate the final result. Most recent studies have not found much difference between the two most common problem decompositions: 1-vs-all, where a classifier is built for each output tag, that distinguishes that tag from all other tags; and 1-vs-1, where a classifier is built for each pair of output tags. Therefore, most people use 1-vs-1, since it is faster to train. (Duan and Keerthi, 2005; Hsu and Lin, 2002)

2.5.3 Mixture Models

Mixture models can be thought of as performing an implicit form of problem subdivision. The motivation for these models comes from the realization that a single parametrized (Gaussian) distribution may not be sufficiently complex to accurately model an underlying distribution. Instead, mixture models assume that the underlying distribution is generated by a generative process where first some class is chosen randomly, and then the output is generated by a per-class distribution. Thus, there is an implicit assumption that the problem is best modelled as being decomposed into a set of sub-problems (the individual distributions). This approach increases performance by replacing a single distribution that has low internal consistency with a small set of distributions that have higher internal consistency; thus, it is related to transformations on output representations that replace a single class tag with a set of more specific tags. However, it differs in that the set of intermediate classes is not explicitly specified or modelled; instead, Expectation Maximization is generally used to pick a set of intermediate classes that maximize the model's accuracy on a training data set. (Alpaydin, 2004; Dasgupta, 1999)

2.6 Prior Work: Output Encodings for Structured Output Tasks

2.6.1 Chunking Representations

The Noun Phrase chunking task was originally formulated as a sequence tagging task in (Ramshaw and Marcus, 1995). Since then, there have been several attempts to improve performance by using different output representations.

The first comparison of the effect of different output encodings on chunking performance was (Tjong Kim Sang and Veenstra, 1999), which adapted a memory-learning NP chunker to use seven different output encodings, including four of the five encodings described in Section 2.2.2 (IOB1, IOB2, IOE1, IOE2) and three encodings that combine the output of two independent learners. Sang & Veenstra found that the IOB1 encoding consistently outperformed the other encodings, although the difference in F-score performance was fairly minor. However, there were more substantial differences in the precision vs recall trade-off, suggesting that the optimal encoding might depend on the relative value of precision and recall in a given task.

This work was built upon by (Tjong Kim Sang, 2000), which used voting to combine the output of five different chunkers, each using a different output encoding. The basic model used for each individual chunker was a memory-based classifier, IB1IG (Daelemans et al., 1999). The five encodings used were `IOB1`, `IOB2`, `IOE1`, `IOE2`, and `IOBES`. Nine different voting methods were tried, but they all yielded similar results, so Sang used the simplest method, majority voting, to present his results. Under this voting method, the best output of each of the five base taggers is converted back into a common encoding (`IOB1`), and then the final encoding tag for each word is chosen individually, using majority voting. Sang evaluated his system on the NP chunking task, and achieved an increase in F-score from 92.8 to 93.26.

(Kudo and Matsumoto, 2001) carried out a similar experiment, but used Support Vector Machines (SVMs) as the underlying model. They used the same five encodings that were used in (Tjong Kim Sang, 2000), but also added a reversed version of each of these encodings, where the system ran backwards through the sentence, rather than forwards. This gave a total of ten basic encodings. They also used a weighted voting scheme, with weights determined by cross-validation. Using this system, they were able to improve performance to 94.22.

(Shen and Sarkar, 2005) also built a voting system based on the five encodings defined by (Tjong Kim Sang, 2000). The model used for the basic chunkers was a second-order HMM, where the output tags were augmented with part of speech and limited lexical information. Voting was performed by converting each of the five taggers' best output back into a common encoding (IOB1), and combining those five tag sequences using majority voting. Shen & Sarkar evaluated their system on NP chunking and CoNNL-2000 data sets. They achieved an increase in F-score on the NP chunking corpus from 94.22 to 95.23. They also pointed out that their model trains much faster than the SVM-based system built by Kudo & Matsumoto.

2.6.2 Semantic Role Representations

Traditionally, the arguments of a verb have been labelled using *thematic roles*, which were first introduced in the mid 1960s (Gruber, 1965; Fillmore, 1968; Jackendoff, 1972). Each thematic role specifies the nature of an argument's relationship with the verb. For example, the agent role specifies that an argument is the active instigator of the verb's action or event. There have been many proposed sets of thematic roles, but there remains little consensus about which set of thematic roles should be used.

Dowty points out that when most traditional thematic role labels are examined closely, they do not appear to be entirely consistent; each of the roles can be subdivided in various ways into more specialized roles (Dowty, 1989). Dowty therefore proposes a weaker definition of thematic roles, where discrete roles are replaced by a set of semantic properties that a role might have (Dowty, 1991). These semantic properties are divided into those which make an argument act more like a "typical agent", and those that make an argument act more like a "typical patient." If an argument has more agent-like properties, it is called a *Proto-Agent*; and if it has more patient-like properties, it is called a *Proto-Patient*.

The difficulty in finding consensus for a single set of thematic roles was one of the motivations behind defining PropBank to use verb-specific roles (Palmer et al., 2005). By defining a separate set of thematic roles for each verb, the PropBank project could avoid the pitfalls of trying to determine when two different verbs' arguments were fulfilling the “same” role, while leaving the door open for future work attempting to do just that. In Chapter 3, I will discuss how a mapping from PropBank to VerbNet was used to replace PropBank’s verb-specific roles with VerbNet’s more general thematic roles, and thereby increase SRL performance.

Modelling SRL

Many researchers have investigated using machine learning for the Semantic Role Labeling task since 2000 (Chen and Rambow, 2003; Gildea and Hockenmaier, 2003; Hacioglu et al., 2003; Moschitti, 2004; Yi and Palmer, 2004; Pradhan et al., 2005b; Punyakanok et al., 2005; Toutanova et al., 2005). For two years, the CoNLL workshop has made this problem the shared task (Carreras and Márquez, 2004; Carreras and Márquez, 2005). Most existing systems use a series of independent classifiers. For example, many systems break the Semantic Role Labeling task into two subtasks, using one classifier to locate the arguments, and a second classifier to assign role labels to those arguments. One disadvantage of using independent classifiers is that it makes it difficult to encode hard and soft constraints between different arguments. For example, these systems can not capture the fact that it is unlikely for a predicate to have two or more agents; or that it is unlikely for a theme (Arg1) argument to precede an agent (Arg0) argument if the predicate uses active voice. Recently, several systems have used methods such as re-ranking and other forms of post-processing to incorporate such dependencies (Gildea and Jurafsky, 2002; Pradhan et al., 2004; Thompson et al., 2003; Sutton and McCallum, 2005; Toutanova et al., 2005).

Transforming SRL Representations

To my knowledge, there is no prior work on applying transformations to SRL representations in order to improve SRL performance.

2.6.3 Parse Tree Representations

Much of the prior research on using output encoding transformations to modify the structure of probabilistic models comes from the parsing community.

Decoupling Tree Structure from Model Structure

The early work on probabilistic parsing focused on PCFGs, which assign a probability to each rule in a CFG, and compute the probability of a parse as the product of the probabilities of the rules used to build it. Mark Johnson points out that this framework assumes that the form of the probabilistic model for a parse tree must exactly match the form of the tree itself (Johnson, 1998). After showing that this assumption can lead to poor models, Johnson suggests that reversible transformations can be used to construct a probabilistic model whose form differs from the form of the desired output tree. He describes four transformations for prepositional-attachment structures, and evaluates those transformations using both a theoretical analysis based on toy training sets, and an empirical analysis based on performance on the Penn Treebank II.

Two of these transformations result in significant improvements to performance: **flatten** and **parent**. The **flatten** transformation replaces select nested tree structures with flat structures, effectively weakening the independence assumptions that are made by the original structure. The **parent** transformation augments each node label with the node label of its parent node, allowing nodes to act as "communication channels" to allow conditional dependency between a node and its grandparent node. Both of these transformations result in a weakening of the model's independence assumptions, while increasing the number of parameters that must be estimated (because they result in a larger set of possible productions). Thus, they can be thought of as an example of the classical "bias versus variance" trade-off. Johnson's empirical results show that, in the case of these two transformations, the reduction in bias overcomes the increase in variance.

Collins' Head-Driven Statistical Parser

In his dissertation, Collins builds on the idea that the structure of a parser's output should be decoupled from the probabilistic model used to generate it (Collins, 1999). In particular, Collins presents a history-based parser that decomposes parse trees into a sequence of "decisions" that preserve specific linguistically motivated lexical and non-lexical dependencies. In Collins' "Model 2" parser, there are four decision types:

1. **Start.** Choose the head-word for the sentence.
2. **Head projection.** Build the spine of a tree.
3. **Subcategorization.** Generate a phrase's complements and adjuncts.
4. **Dependency.** Choose the head word for a complement or an adjunct.

Although Collins describes his parser in terms of a history-based sequence of decisions, it can also be thought of as a complex tree transformation. In particular, Figure 2.13 gives an example showing how Collins' "Model 2" parser can be expressed as a transformation from the canonical Treebank-style encoding to a new encoding that introduces additional structure. Each node in this transformed tree corresponds to a decision variable in Collins' model. Under this transformed encoding, Collins' "Model 2" parser can implemented as a PCFG.[6]

Collins argues that two particularly important criteria for deciding how to decompose a structured problem are discriminative power and compactness. The *discriminative power* of a decomposition reflects whether its local subproblems' parameters include enough contextual information to accurately choose the correct decision. Collins points out that simple PCFGs fail in this respect, because they are insensitive to lexical and structural contextual information that is necessary to make correct

[6] Collins' model makes use of linear interpolated backoff to reduce the adverse effect of data sparsity. In order to accurately implement Collins' parser, the PCFG would need to implement these backoff methods, along with a number of additional transformations that have been glossed over here. See (Collins, 1999) and (Bikel, 2004b) for a more detailed account.

local decisions. The *compactness* of a decomposition measures the number of free parameters that must be estimated. The more parameters a model has, the more training data will be required to accurately train those parameters. Thus, given two models with equal discriminative power, we should prefer the more compact model.

In order to ensure that a model has sufficient discriminative power, Collins suggests that the notion of *locality* should be used to determine what the dependencies should be between local subproblems. In particular, the decomposition should preserve structural connections between any variables that are within each others' domain of locality. As was discussed in Section 2.1.2, Collins argues that this notion of locality is a generalization of the notion of causality from work on Bayesian Networks.

Analysis of Collins' Parser

Daniel Bikel provides a detailed analysis of Collins' parser, which provides some insight into which aspects of its decomposition are most beneficial to performance (Bikel, 2004b). This analysis is based upon a flexible re-implementation of Collins' parser, which can be used to turn various features of Collins' parser on and off, and to tweak them in different ways. Bikel evaluates the impact of individual features of Collins' parser by looking at how performance changes when those features are turned off in different combinations.

Bikel begins by describing a large number of previously unpublished details. Although these details have a significant joint effect on the parser's performance (11% error reduction), their individual contributions are relatively small.

He then analyzes the effect of three features thought to be important to the performance of Collins' parser: bi-lexical dependencies, choice of lexical head words, and lexico-structural dependencies. Somewhat surprisingly, he finds that the performance drop caused by omitting bi-lexical dependencies is relatively minor. He explains this small drop by showing that the bi-lexical dependencies seen in a new

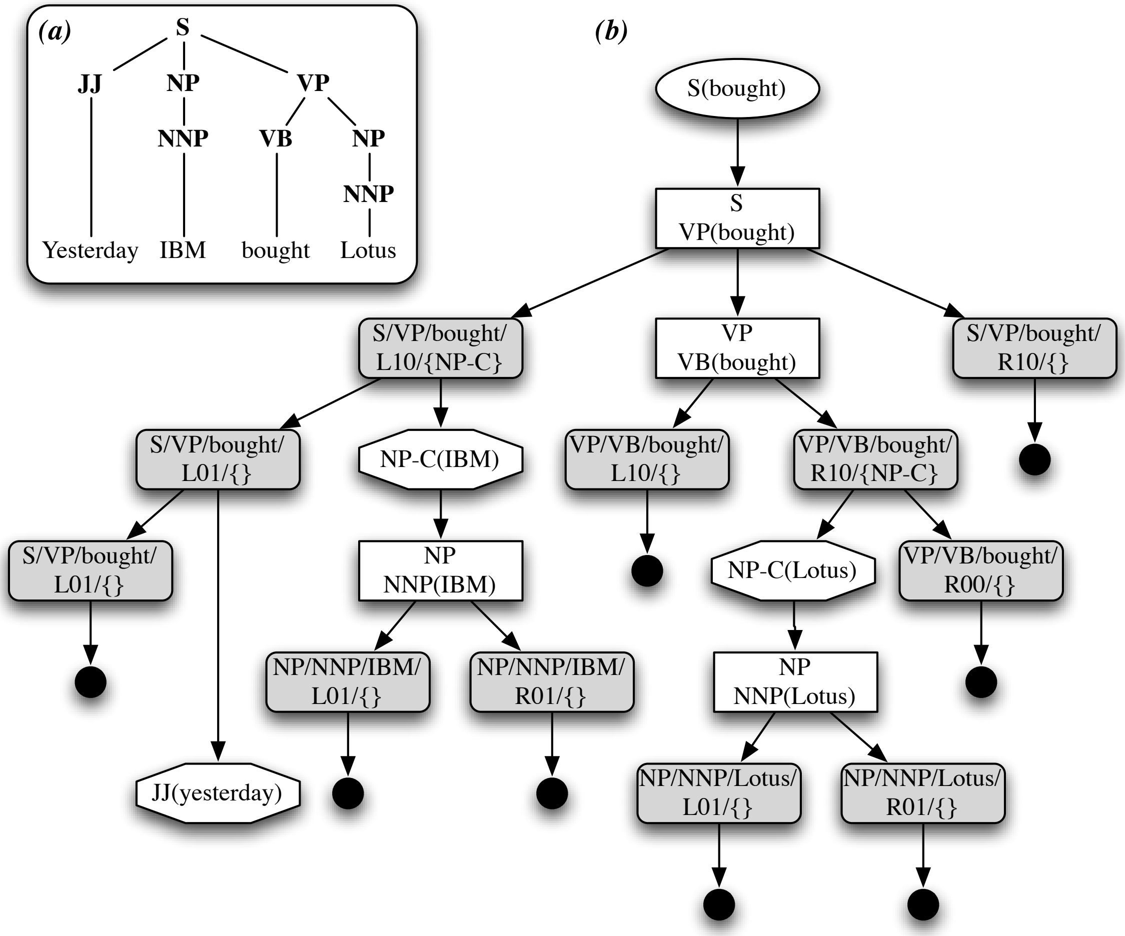


Figure 2.13: **Collins' "Model 2" Parser as Tree Transformation**. This figure illustrates how Collins' Parser can be modelled using the notion of encoding transformation. The structure *(a)* shows the canonical parse tree encoding for a simple example sentence. The structure *(b)* shows an encoding of the same parse tree that reflects Collins' choice of decomposition for the parsing problem. The elliptical node "S(bought)" corresponds to the **start** decision, and consists of a phrase label ("S") and a head word ("bought"). The square white nodes correspond to **head projection** decisions; each contains the phrase label, headword, and parent's phrase label for a single constituent. The shaded nodes correspond to **subcategorization** decisions; each contains a phrase label, a parent phrase label, a headword, a direction, a distance metric, and a set of sub-categorized arguments. The black circle nodes represent `STOP` tokens for the sub-categorization frames. The octagonal white nodes correspond to **dependency** decisions, and select head words for complements and adjuncts. See (Collins, 1999) and (Bikel, 2004b) for more information about Collins' parser.

sentence are almost never present in the training corpus; in other words, the training corpus is too small for these very sparse features to be much help. Bikel also finds that the the head-choice heuristics do not have a major impact on performance. However, he finds that the use of lexico-structural dependencies (i.e., dependencies between a lexical word and a structural configuration) *is* quite important. Unlike bi-lexical dependencies, these lexico-structural dependencies are associated with enough training data to make them useful for evaluating novel sentences. And as has been shown before, lexical information is often important in making structural decisions, such as the decision of whether a prepositional phrase should attach at the verb phrase or noun phrase level.

Splitting States to Improve Unlexicalized Parsing

Although the introduction of lexico-structural dependencies is clearly very important to the performance of advanced lexicalized parsers, they are by no means the only reason that these parsers out-perform naive PCFG parsers. In order to explore which non-lexical dependencies are important to improving parser performance, Klein & Manning applied a manual hill-climbing approach to develop a sequence of tree transformations that improve upon the performance of a baseline PCFG system (Klein and Manning, 2003a). Using this method, they find a sequence of 17 transformations that increases the performance of their unlexicalized parser to a level comparable to that of basic lexicalized parsers.

Their baseline system differs from a simple PCFG in that it begins by decomposing all nodes with a branching factor greater than 2 into binary branching nodes. This binary branching decomposition is centered on the head node; and new node labels are created for the intermediate nodes. These new node labels, which Klein & Manning refer to as "intermediate symbols," initially consist of the original node label plus the part of speech of the head word; but they may be modified by transformation operations, as described below.

All of Klein & Manning's transformations consist of splitting select node labels into two or more specialized labels. The first two transformations relax the conditional independence assumptions of the simple PCFG model by adding contextual information about a node's parents or siblings to that node's label. The first of these transformations, **vertical-markovization**(n), augments each non-intermediate node label with the labels of its n closest ancestor nodes. This is essentially a generalization of Mark Johnson's **parent** transformation. The second transformation, **horizontal-markovization**(n), is analogous, except that it applies to intermediate nodes, and thus adds information about siblings instead of ancestors. Klein & Manning also consider a variant of these transformations which does not split intermediate states that occur 10 or fewer times in the training corpus. For their overall system, they settle on a value of $n = 2$ for both Markovization transformations.

Klein & Manning describe fifteen additional transformations, which split node labels based on a variety of contextual features, including both "internal context" (features of the phrase itself) and "external context" (features of the tree outside the phrase). Individually, these transformations improve F_1 performance by between 0.17% and 2.52%; in total, performance is improved by 14.4%, from 72.62% to 87.04%.

A Factored Parsing Model

Klein and Manning describe a novel model for parsing that combines two different encodings for the parse tree: a simple PCFG, and a dependency structure (Klein and Manning, 2003c; Klein and Manning, 2003b). These two encodings are modelled independently, and then their probabilities are combined by simple multiplication. In other words, if T is a tree, and τ_{PCFG} and τ_{dep} are encoding functions mapping trees to PCFGs and dependency structures respectively, then Klein and Manning

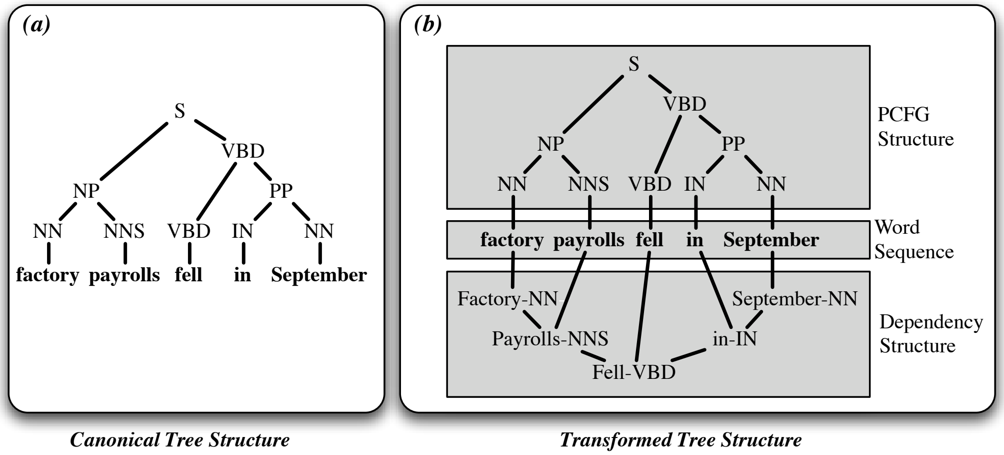


Figure 2.14: **Klein & Manning's Factored Model as a Tree Transformation**. Klein and Manning's factored parsing model $P(T, D) = P(T)P(D)$ can be thought of as a tree transformation that replaces the canonical structure *(a)* with a new structure *(b)* that describes the sentence's structure using two separate (disconnected) pieces: one describing the sentence's PCFG structure, and the other describing its dependency structure.

model the probability of a tree T as:

$$P(T) = P\left(\tau_{PCFG}(T)\right) P\left(\tau_{dep}(T)\right) \tag{2.44}$$

This decomposition is consistent with the common psycholinguistic belief that syntax and lexical semantics are two relatively decoupled modules, with syntax responsible for constraining the set of acceptable structural configurations independent of individual lexical items, and lexical semantics responsible for resolving ambiguities. Figure 2.14 illustrates how this factored model can be represented as an output encoding transformation.

As Klein and Manning point out, this decomposition assigns probability mass to invalid output structures. In particular, since the two sub-models are entirely independent, there is nothing to prevent them from building structures with different terminal strings. Klein and Manning suggest that this problem could be alleviated by discarding all inconsistent outputs, and re-normalizing the remaining probabilities to sum to one. However, a more principled solution might be switching from generative models to conditional models. In particular, Equation 2.44 could be replaced by the following conditional variant, where S is the input sentence:

$$P(T|S) = P\left(\tau_{PCFG}(T|S)\right) P\left(\tau_{dep}(T|S)\right) \tag{2.45}$$

Since both models are conditioned on S, they can no longer generate incompatible terminal strings.[7]

Using their factored model, Klein and Manning show that it is possible to perform efficient exact search using an A* parser. The A* algorithm provides guidance to a search problem by making use of an estimate of the cost of completing a given search path. If this estimate provides a lower bound on the cost, then the A* algorithm is guaranteed to find the optimal search path. In the context of bottom-up parsing, search paths correspond to phrase structures, and the cost of completing a search path is inversely related to the maximal "outside probability" of a given phrase structure α:

$$P_{outside}(\alpha) = \max_{T:\alpha\in T} P(T) - P(\alpha) \tag{2.46}$$

Because the two factored models proposed by Klein and Manning are individually relatively simple, it is possible to calculate the outside probability for these individual models analytically. These two outside probabilities can then be combined to form an estimate of the outside probability in the joint model by simply multiplying them:

$$P_{outside}(\alpha) \leq P_{outside}\left(\tau_{PCFG}(\alpha)\right) P_{outside}\left(\tau_{dep}(\alpha)\right) \tag{2.47}$$

Using this estimate for the outside probability, Klein and Manning show that an A* parser using their factored model performs comparably to existing lexicalized parsers that use a joint model to learn lexical and structural preferences.

[7] This move to conditional models solves the problem of incompatible terminal strings, but applying the two models independently may still generate incompatible structures. In particular, dependency structures impose constraints on the set of possible phrase bracketings; and those constraints are not always compatible with all possible PCFG trees. This issue could be addressed by the renormalization trick proposed by Klein and Manning, or by adding a limited set of dependencies between the two models.

Automatic State Splitting: PCFG with Latent Annotations

The approaches discussed thus far improve parsing performance over simple PCFGs by applying problem decompositions that do not directly follow the structure of the parse tree. Each of these approaches uses a fixed decomposition, motivated by a combination of theoretical considerations and trial-and-error. Matsuzaki, Miyao, & Tsujii examine the possibility of automating the task of choosing an optimal problem decomposition (Matsuzaki et al., 2005). They restrict their attention to the class of problem decompositions that is formed by augmenting PCFG nodes with discrete feature values (or *latent annotations*). These decompositions effectively transform the canonical parse tree by subdividing the existing phrase types (NP, PP, etc) into sub-types.

This transformation differs from most of the transformations discussed so far in that it does not define a one-to-one mapping between canonical values and transformed values. In particular, if n discrete feature values are used to augment canonical trees, then a canonical tree with m nonterminal nodes corresponds to n^m different augmented trees (one for each possible assignment of feature values to nodes). As a result, applying standard parsing algorithms to the augmented PCFG will generate the most likely annotated tree; but this does not necessarily correspond to the most likely unannotated (canonical) tree. Matsuzaki, Miyao, & Tsujii therefore explore the use of three different variants on the CKY parsing algorithm which approximate the search for the best unannotated tree.

Starting with a PCFG grammar and a fixed set of feature values, Matsuzaki, Miyao, & Tsujii apply the Expectation Maximization algorithm to iteratively improve upon the PCFG's transition probabilities. As a result, the PCFG automatically learns to make use of the feature values in such a way that the likelihood of the training corpus is maximized. Using their approximate-best parsing algorithms on the PCFG generated by EM, Matsuzaki, Miyao, & Tsujii's parser achieves performance comparable to unlexicalized parsers that make use of hand-crafted problem decompositions.

Automatic State Splitting: Splitting Individual Nodes

(Petrov et al., 2006) uses an automatic approach to tree annotation that is similar to the approach taken by Matsuzaki, Miyao, & Tsujii. But their approach differs from the approach taken by Matsuzaki et al. in that they split various nonterminals to different degrees, as appropriate to the actual complexity in the data. For example, their system finds that the preposition phrase tag (PP) should be split into 28 distinct categories, while just 2 categories are sufficient to model conjunction phrases (CONJP).

Another important difference between their system and the PCFG-LA system described by Matsuzaki et al. is that node decompositions are performed incrementally, via binary splits. This incremental approach gives rise to a tree of node labels which are much more amenable to linguistic interpretation than the categories generated by the PCFG-LA system.

The learning algorithm for this system begins with the original set of PCFG labels. It then iteratively performs three steps: **split**, **EM**, and **merge**. The **split** step divides each node label into two new labels; and divides the probability mass of the associated PCFG productions between these new labels. In order to break the symmetry between the new labels, a small amount of randomness is added to the PCFG production probabilities. The **EM** step uses Expectation Maximization to learn probabilities for all rules by optimizing the likelihood of the training data. The **merge** step then examines each split that was made, and estimates what the effect would be of removing the split. If the effect is small enough, then the two split nodes are merged back together. This merge operation can be thought of as analogous to the pruning step in the construction of decision trees, where decision structures that do not significantly improve performance are pruned away to reduce the number of parameters that the model must learn, thereby avoiding over-fitting.

This split-merge procedure is used because it is much easier to estimate what the effect of a merge will be than it is to estimate what the effect of a split will be.

Like the PCFG-LA system, this system does not define a one-to-one mapping between canonical values and transformed values: a single canonical tree will correspond to a relatively large set of annotated trees. As a result, calculating the best unannotated tree for a given sentence is NP-hard. Petrov et al. therefore perform parsing using an algorithm that maximizes the total number of correct productions, rather than the probability of the unannotated parse.

Chapter 3

Improving Class Coherence by Transforming Semantic Role Labels via SemLink

Semantic role labeling involves locating the arguments of a verb, and assigning them role labels that describe their semantic relationship with the verb. However, there is still little consensus in the linguistic and NLP communities about what set of role labels is most appropriate. The Proposition Bank (or "PropBank") corpus avoids this issue by using theory-agnostic labels (Arg0, Arg1, . . . , Arg5), and by defining those labels to have verb-specific meanings (Palmer et al., 2005). Under this scheme, PropBank can avoid making any claims about how any one verb's arguments relate to other verbs' arguments, or about general distinctions between verb arguments and adjuncts.

However, there are several limitations to this approach. The first is that it can be difficult to make inferences and generalizations based on role labels that are only meaningful with respect to a single verb. Since each role label is verb-specific, we can not confidently determine when two different verbs' arguments have the same role; and since no unique meaning is associated with each tag, we can not make generalizations across verb classes. In contrast, the use of a shared set of role labels, such as thematic roles, would facilitate both inferencing and generalization.

The second issue with PropBank's verb-specific approach is that it can make training automatic semantic role labeling (SRL) systems more difficult. A vast amount of data would be needed to train the verb-specific models that are theoretically mandated by PropBank's design. Instead, researchers typically build a single model for each numbered argument (Arg0, Arg1, . . . , Arg5). This approach works surprisingly well, mainly because an explicit effort was made when PropBank was created to use arguments Arg0 and Arg1 consistently across different verbs; and because those two argument labels account for 85% of all arguments. However, this approach causes the system to conflate different argument types, especially with the highly overloaded arguments Arg2-Arg5. As a result, these argument labels are quite difficult to learn.

A final difficulty with PropBank's current approach is that it limits SRL system robustness in the face of verb senses and verb constructions that were not included in the training data (namely, the Wall Street Journal). If a PropBank-trained SRL system encounters a novel verb or verb usage, then there is no way for it to know which role labels are used for which argument types, since role labels are defined so specifically. For example, even if there is ample evidence that an argument is serving as the destination for a verb, an SRL system trained on PropBank will be unable to decide which numbered argument (Arg0-5) should be used for that particular verb unless it has seen that verb used with a destination argument in the training data. This type of problem can happen quite frequently when SRL systems are run on novel genres, as reflected in the relatively poor performance of most state-of-the-art SRL systems when tested on a novel genre, the Brown corpus, during CoNLL 2005. For example, the SRL system described in (Pradhan et al., 2005b; Pradhan et al., 2005a) achieves an F-score of 81% when tested on the same genre as it is trained on (WSJ); but that score drops to 68.5% when the same system is tested on a different genre (the Brown corpus). DARPA-GALE is funding an ongoing effort to annotate additional genres with PropBank information, but better techniques for generalizing the semantic role labeling task are still needed.

To help address these three difficulties, we have constructed a mapping between PropBank and another lexical resource, VerbNet. By taking advantage of VerbNet's more consistent and coherent set of labels, we can generate more useful role label annotations with a resulting improvement in SRL performance, especially for novel genres.

3.1 VerbNet

VerbNet (Schuler, 2005) consists of hierarchically arranged verb classes, inspired by and extended from classes of Levin 1993 (Levin, 1993). Each class and subclass is characterized extensionally by its set of verbs, and intensionally by a list of the arguments of those verbs and syntactic and semantic information about the verbs. The argument list consists of thematic roles (23 in total) and possible selectional restrictions on the arguments expressed using binary predicates. The syntactic information maps the list of thematic arguments to deep-syntactic arguments (i.e., normalized for voice alternations, and transformations). The semantic predicates describe the participants during various stages of the event described by the syntactic frame.

The same thematic role can occur in different classes, where it will appear in different predicates, providing a class-specific interpretation of the role. VerbNet has been extended from the original Levin classes, and now covers 4526 senses for 3769 verbs. A primary emphasis for VerbNet is the grouping of verbs into classes that have a coherent syntactic and semantic characterization, that will eventually facilitate the acquisition of new class members based on observable syntactic and semantic behavior. The hierarchical structure and small number of thematic roles is aimed at supporting generalizations.

3.2 SemLink: Mapping PropBank to VerbNet

Because PropBank includes a large corpus of manually annotated predicate-argument data, it can be used to train supervised machine learning algorithms, which can in turn provide PropBank-style annotations for novel or unseen text. However, PropBank's verb-specific role labels are somewhat problematic. Furthermore, PropBank lacks much of the information that is contained in VerbNet, including information about selectional restrictions, verb semantics, and inter-verb relationships.

Therefore, as part of the SemLink project, we have created a mapping between VerbNet and PropBank (Loper et al., 2007), which will allow us to use the machine learning techniques that have been developed for PropBank annotations to generate more semantically abstract VerbNet representations. Additionally, the mapping can be used to translate PropBank-style numbered arguments (Arg0. . . Arg5) to VerbNet thematic roles (Agent, Patient, Theme, etc.), which should allow us to overcome the verb-specific nature of PropBank.

The SemLink mapping between VerbNet and PropBank consists of two parts: a *lexical mapping* and an *instance classifier*. The lexical mapping is responsible for specifying the potential mappings between PropBank and VerbNet for a given word; but it does not specify which of those mappings should be used for any given occurrence of the word. That is the job of the instance classifier, which looks at the word in context, and decides which of the mappings is most appropriate. In essence, the instance classifier is performing word sense disambiguation, deciding which lexeme from each database is correct for a given occurrence of a word. In order to train the instance classifier, we semi-automatically annotated each verb in the PropBank corpus with VerbNet class information.[1] This *mapped corpus* was then used to build the instance classifier. More details about the mapping, and how it was created, can be found in (Loper et al., 2007).

[1] Excepting verbs whose senses are not present in VerbNet (24.5% of instances).

3.3 Analysis of the Mapping

An analysis of the mapping from PropBank role labels to VerbNet thematic roles confirms the belief that PropBank roles Arg0 and Arg1 are relatively coherent, while roles Arg2-5 are much more overloaded. Table 3.1 shows how often each PropBank role was mapped to each VerbNet thematic role, calculated as a fraction of instances in the mapped corpus. From this figure, we can see that Arg0 maps to agent-like roles, such as "agent" and "experiencer," over 94% of the time; and Arg1 maps to patient-like roles, including "theme," "topic," and "patient," over 82% of the time. In contrast, arguments Arg2-5 get mapped to a much broader variety of roles. It is also worth noting that the sample size for arguments Arg3-5 is quite small in comparison with arguments Arg0-2, suggesting that any automatically built classifier for arguments Arg3-5 will suffer severe sparse data problems for those arguments.

3.4 Training an SRL system with VerbNet Roles to Achieve Robustness

An important issue for state-of-the-art automatic SRL systems is robustness: although they receive high performance scores when tested on the Wall Street Journal (WSJ) corpus, that performance drops significantly when the same systems are tested on a corpus from another genre. This performance drop reflects the fact that the WSJ corpus is highly specialized, and tends to use genre-specific word senses for many verbs. The 2005 CoNLL shared task has addressed this issue of robustness by evaluating participating systems on a test set extracted from the Brown corpus, which is very different from the WSJ corpus that was used for training. The results suggest that there is much work to be done in order to improve system robustness.

One of the reasons that current SRL systems have difficulty deciding which role label to assign to a given argument is that role labels are defined on a per-verb basis.

Arg0	(n=45,579)
Agent	85.4%
Experiencer	7.2%
Theme	2.1%
Cause	1.9%
Actor1	1.8%
Theme1	0.8%
Patient1	0.2%
Location	0.2%
Theme2	0.2%
Product	0.1%
Patient	0.0%
Attribute	0.0%

Arg1	(n=59,884)
Theme	47.0%
Topic	23.0%
Patient	10.8%
Product	2.9%
Predicate	2.5%
Patient1	2.4%
Stimulus	2.0%
Experiencer	1.9%
Cause	1.8%
Destination	0.9%
Theme2	0.7%
Location	0.7%
Source	0.7%
Theme1	0.6%
Actor2	0.6%
Recipient	0.5%
Agent	0.4%
Attribute	0.2%
Asset	0.2%
Patient2	0.2%
Material	0.2%
Beneficiary	0.0%

Arg2	(n=11,077)
Recipient	22.3%
Extent	14.7%
Predicate	13.4%
Destination	8.6%
Attribute	7.6%
Location	6.5%
Theme	5.5%
Patient2	5.3%
Source	5.2%
Topic	3.1%
Theme2	2.5%
Product	1.5%
Cause	1.2%
Material	0.8%
Instrument	0.6%
Beneficiary	0.5%
Experiencer	0.3%
Actor2	0.2%
Asset	0.0%
Theme1	0.0%

Arg3	(n=609)
Asset	38.6%
Source	25.1%
Beneficiary	10.7%
Cause	9.7%
Predicate	9.0%
Location	2.0%
Material	1.8%
Theme1	1.6%
Theme	0.8%
Destination	0.3%
Instrument	0.3%

Arg4	(n=18)
Beneficiary	61.1%
Product	33.3%
Location	5.6%

Arg5	(n=17)
Location	100.0%

Table 3.1: **PropBank Role Mapping Frequencies**. This table lists the frequency with which each PropBank numbered argument is mapped to each VerbNet thematic role in the mapped corpus. The number next to each PropBank argument (n) indicates the number of occurrences of that numbered argument in the mapped corpus.

This is less problematic for Arg0 and Arg1, where a conscious effort was made to be consistent across verbs; but is a significant problem for Args[2-5], which tend to have very verb-specific meanings. This problem is exacerbated even further on novel genres, where SRL systems are more likely to encounter unseen verbs and uses of arguments that were not encountered in the training data.

3.4.1 Addressing Current SRL Problems via Lexical Mappings

By exploiting the mapping between PropBank and VerbNet, we can transform the data to make it more consistent. In particular, we can use the mapping to transform the verb-specific PropBank role labels into the more general thematic role labels that are used by VerbNet. Unlike the PropBank labels, the VerbNet labels are defined consistently across verbs; and therefore it should be easier for statistical SRL systems to model them. Furthermore, since the VerbNet role labels are significantly less verb-dependent than the PropBank roles, the SRL's models should generalize better to novel verbs, and to novel uses of known verbs.

3.5 Experiments

Section 3.6 describes joint work done with Szu-ting Yi and Martha Palmer, in which we performed several preliminary experiments to verify the feasibility of performing semantic role labeling with VerbNet thematic roles (Loper et al., 2007; Yi et al., 2007). Section 3.7 describes a set of experiments looking at the use of "subset features," which allow the SRL model to avoid some of the sparse data problems that were encountered in the preliminary work. Section 3.8 describes experiments that assume that VerbNet-style thematic roles are a more useful output for SRL; and compare the effectiveness of performing argument mapping before argument classification with the effectiveness of performing argument mapping after argument classification. Section 3.9 describes experiments that compare the SemLink mapping with a mapping that was hand-generated based the argument descriptions provided by the PropBank frames.

3.6 Preliminary SRL Experiments on SemLink

To verify the feasibility of performing semantic role labeling with VerbNet thematic roles, we used the SemLink mapping to transform the PropBank corpus in several different ways, and adapted Szu-ting Yi's Semantic Role Labeling system to use these transformed corpora as training data.

3.6.1 The Baseline SRL System

Szu-ting Yi's SRL system is a Maximum Entropy based pipelined system which consists of four components: Pre-processing, Argument Identification, Argument Classification, and Post Processing. The Pre-processing component pipes a sentence through a syntactic parser and filters out constituents which are unlikely to be semantic arguments based on their location in the parse tree. The Argument Identification component is a binary MaxEnt classifier, which tags candidate constituents as arguments or non-arguments. The Argument Classification component is a multi-class MaxEnt classifier which assigns a semantic role to each constituent. The Post Processing component further selects the final arguments based on global constraints. Our experiments mainly focused on changes to the Argument Classification stage of the SRL pipeline, and in particular, on changes to the set of output tags. For more information on Szu-ting Yi's original SRL system, including information about the feature sets used for each component, see (Yi and Palmer, 2004; Yi and Palmer, 2005).

Group 1	Group 2	Group 3	Group 4	Group 5
Theme	Source	Patient	Agent	Topic
Theme1	Location	Product	Actor2	
Theme2	Destination	Patient1	Experiencer	Group 6
Predicate	Recipient	Patient2	Cause	Asset
Stimulus	Beneficiary			
Attribute	Material			

Figure 3.1: **Thematic Role Grouping A**. This grouping of thematic roles was used for subdividing Arg1 in Experiment 3.6.2. Karin Kipper assisted in creating the groupings.

3.6.2 Applying the SemLink Mapping to Individual Arguments

We conducted two sets of experiments to test the effect of applying the SemLink mapping to individual arguments. The first set used the mapping to subdivide Arg1; and the second set used the mapping to subdivide Arg2. Since Arg2 is used in very verb-dependent ways, we expect that mapping it to VerbNet role labels will increase our performance. However, since a conscious effort was made to keep the meaning of Arg1 consistent across verbs, we expect that mapping it to VerbNet labels will provide less of an improvement.

Each experiment compares two SRL systems: one trained using the original PropBank role labels; the other trained with the argument role under consideration (Arg1 or Arg2) subdivided based on which VerbNet role label it maps to.

We found that subdividing directly into individual role labels created a significant sparse data problem, since the number of output tags was increased from 6 to 28. We therefore manually grouped the VerbNet thematic roles into coherent groups of similar thematic roles, shown in Figure 3.1 (for the Arg1 experiments) and Figure 3.2 (for the Arg2 experiments). Thus, for the Arg1 experiments, the transformed output tags were {Arg0, $\text{Arg1}_{\text{group1}}$, ..., $\text{Arg1}_{\text{group5}}$, Arg2, Arg3, Arg4, Arg5, ArgM}; and for the Arg2 experiments, the transformed output tags were {Arg0, Arg1, $\text{Arg2}_{\text{group1}}$, ..., $\text{Arg2}_{\text{group6}}$, Arg4, Arg4, Arg5, ArgM}.

Group 1	Group 2	Group 3	Group 4	Group 5
Recipient Destination Location Source Material Beneficiary	Extent Asset	Predicate Attribute Theme Theme1 Theme2 Topic	Patient2 Product	Instrument Cause Experiencer Actor2

Figure 3.2: **Thematic Role Grouping B**. This grouping of thematic roles was used for subdividing Arg2 in Experiment 3.6.2. Karin Kipper assisted in creating the groupings.

The training data for both experiments is the portion of Penn Treebank II (sections 02-21) that is covered by the mapping. We evaluated each experimental system using two test sets: section 23 of the Penn Treebank II, which represents the same genre as the training data; and the PropBank-annotated portion of the Brown corpus, which represents a very different genre. For the purposes of evaluation, the experimental systems' subdivided roles Arg$n_{\mathrm{group}i}$ were simply treated as members of Arg$n$. This was necessary to allow direct comparison between the baseline system and the experimental systems; and because no gold-standard data is available for the subdivided roles in the Brown Corpus.

Results and Discussion

Table 3.2 gives the results of the mapping on SRL overall performance, tested on the WSJ corpus Section 23; Table 3.3 shows the effect on SRL overall system performance, tested on the Brown corpus. Systems Arg1-Original and Arg2-Original are trained using the original PropBank labels, and show the baseline performance of our SRL system. Systems Arg1-Mapped and Arg2-Mapped are trained using PropBank labels augmented with VerbNet thematic role groups. As mentioned above, system performance was evaluated based solely on the PropBank role labels (and not the subdivided labels) in order to allow direct comparison between the original system and the mapped systems.

We had hypothesized that with the use of thematic roles, we would be able to

System	**Precision**	**Recall**	F_1
Arg1-Original	89.24	77.32	82.85
Arg1-Mapped	90.00	76.35	82.61
Arg2-Original	73.04	57.44	64.31
Arg2-Mapped	84.11	60.55	70.41

Table 3.2: **Results from Experiment 3.6.2 (WSJ Corpus)**. SRL System Performance on Arg1 Mapping and Arg2 Mapping, tested using the *WSJ corpus (section 23)*. This represents performance on the same genre as the training corpus.

System	**Precision**	**Recall**	F_1
Arg1-Original	86.01	71.46	78.07
Arg1-Mapped	88.24	71.15	78.78
Arg2-Original	66.74	52.22	58.59
Arg2-Mapped	81.45	58.45	68.06

Table 3.3: **Results from Experiment 3.6.2 (Brown Corpus)**. SRL System Performance on Arg1 Mapping and Arg2 Mapping, tested using the *PropBank-annotated portion of the Brown corpus*. This represents performance on a different genre from the training corpus.

create a more consistent training data set which would result in an improvement in system performance. In addition, the thematic roles would behave more consistently than the overloaded Args[2-5] across verbs, which should enhance robustness. However, since in practice we are also increasing the number of argument labels an SRL system needs to tag, the system might suffer from data sparseness. Our hope was that the enhancement gained from the mapping will outweigh the loss due to date sparseness.

From Table 3.2 and Table 3.3 we see the F_1 scores of Arg1-Original and Arg1-Mapped are not statistically different on both the WSJ corpus and the Brown corpus. These results confirm the observation that Arg1 in the PropBank behaves fairly verb-independently so that the VerbNet mapping does not provide much benefit. The increase of precision due to a more coherent training data set is compensated for by the loss of recall due to data sparseness.

The results of the Arg2 experiments tell a different story. Both precision and recall are improved significantly, which demonstrates that the Arg2 label in the PropBank is quite overloaded. The Arg2 mapping improves the overall results (F_1) on the WSJ by 6% and on the Brown corpus by almost 10%. As a more diverse corpus, the Brown corpus provides many more opportunities for generalizing to new usages. Our new SRL system handles these cases more robustly, demonstrating the consistency and usefulness of the thematic role categories.

3.6.3 Improved Argument Distinction via Mapping

The ARG2-Mapped system generalizes well both on the WSJ corpus and the Brown corpus. In order to explore the improved robustness brought by the mapping, we extracted and observed the 1,539 instances to which the system ARG2-Mapped assigned the correct semantic role label, but which the system ARG2-Original failed to predict. From the confusion matrix depicted in Table 3.4, we discover the following:

The mapping makes ARG2 more clearly defined, and as a result there is a better distinction between ARG2 and other argument labels: Among the 1,539 instances that ARG2-Original didn't tag correctly, 233 instances are not assigned an argument label, and 1,252 instances of ARG2-Original confuse the ARG2 label with another argument label: the system ARG2-Original assigned the ARG2 label to 50 ARG0's, 716 ARG1's, 1 ARG3 and 482 ARGM's, and assigned other argument labels to 3 ARG2's.

3.7 Applying the SemLink Mapping with Subset Features

As was noted in Section 3.6.2, we found that subdividing each of the PropBank argument roles Arg0-Arg5 into the 28 thematic roles used by VerbNet caused significant sparse data problems. In particular, by splitting the argument labels, we

Confusion Matrix		ARG2-Original		
		ARG1	ARG2	ARGM
ARG2-Mapped	ARG0	53	50	-
	ARG1	-	716	-
	ARG2	1	-	2
	ARG3	-	1	-
	ARGM	1	482	-
233 ARG2-Mapped arguments are not labeled by ARG2-Original				

Table 3.4: **Confusion Matrix for Experiment 3.6.2**. Confusion matrix on the 1,539 instances which ARG2-Mapped tags correctly and ARG2-Original fails to predict.

significantly reduce the amount of training data that is available for to model each label.

3.7.1 Partially Subdividing Labels

One solution to this problem would be to use linear interpolated backoff between different models. In particular, we could use averaging to combine the results of the original (non-subdivided) system with the new (subdivided) system. But the conditional exponential models used by the Maximum Entropy learning method provide us with a more principled solution to the problem, by allowing us to define features that apply to more than one label. In a conditional exponential model, each feature is defined as a function of both the input and the output. Typically, features have the following form:

$$f_i(x,y) = \begin{cases} 1 & \text{if } g_i(x) = 1 \text{ and } y = l_i \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

In other words, feature f_i is true if the input satisfies some context function (g_i) and if the output label is some constant (l_i). We will refer to features with this form as "simple features."

The amount of training data on which the value of each feature's weight can be based is determined by the number of times that feature fires. For simple features, this is equal to the number of times a feature's context function occurs with a specific label. Thus, increasing the size of the label set decreases the amount of training data that can be used to train each feature.

We can get around this problem by adding features that have a slightly more general form:

$$f_i(x, y) = \begin{cases} 1 & \text{if } g_i(x) = 1 \text{ and } y \in L_i \\ 0 & \text{otherwise} \end{cases} \tag{3.2}$$

Features of the form shown in 3.2 are true if the input satisfies some context function (g_i) and if the output label is an element of a fixed set (L_i). We will refer to features with this form as "subset features."

Subset features can be used to learn the predictive relationship between a context function and a set of related labels. Since these features can use training data from examples that have any label in L_i, they are more robust against sparse data problems than simple features, especially when using large label sets.

By using a model that contains both simple and subset features, we allow the maximum entropy learning algorithm to make generalizations over sets of labels where possible (using subset features), and to learn more specific predictive relationships between context functions and labels when there is enough training data to support them (using simple features).

In order to prevent the model from over-fitting, it is important to provide an appropriate Gaussian prior. This prior ensures that preference is given to features that are supported by more training data; and that features that are supported by less training data are only used when they are highly predictive. In general, this encourages the model to make more use of the subset features, and to only use the simple features when they make predictions that are not already made by the subset features.

3.7.2 Subset Features for the SemLink Mapping

In order to test the effectiveness of subset features at overcoming the sparse data problems we encountered in (Loper et al., 2007) and (Yi et al., 2007), I built a series of argument classifiers whose labels consist of pairs of PropBank labels (Arg0-5) and VerbNet labels (Agent, Patient, etc.)[2]. Typical examples of labels are $\text{Arg0}_{\text{agent}}$ and $\text{Arg2}_{\text{recipient}}$. In addition to these paired labels, the model included the traditional ArgM and ArgA labels (e.g., ArgM-LOC).

In addition to the simple features based on individual argument labels, I defined two groups of subset features. The first group, `PropBankSubsets`, uses label sets that pair a single PropBank label with any VerbNet label (e.g. Arg0_*). This group allows models to learn generalizations that apply to examples with a given PropBank label. The second group, `VerbNetSubsets`, uses label sets that pair a single VerbNet label with any numbered PropBank label (e.g., $\text{Arg*}_{\text{agent}}$). This group allows models to learn generalizations that apply to examples with a given VerbNet label.

Because the MaxEnt classifier used by Szu-ting Yi's SRL system did not provide the necessary support to use subset features, I modified it to use the MEGA Model Optimization Package (Daumé III, 2004) instead.

To evaluate the effectiveness of using subdivided labels and subset features, I compared the performance of five systems:

- `PropBank`: Uses only the `PropBankSubset` features. That this is exactly equivalent to training a model using the original (PropBank) label set. It is therefore treated as the baseline system.

- `Simple`: Uses only the simple features.

- `Simple+PropBank`: Uses both the simple features and the `PropBankSubset` features.

[2] I collapsed "Actor" and "Actor1" into a single role; and similarly for "Patient" and "Theme."

- `Simple+VerbNet`: Uses both the simple features and the `VerbNetSubset` features.

- `Simple+PropBank+VerbNet`: Uses the simple features, the `PropBankSubset` features, and the `VerbNetSubset` features.

In all cases, the training data is the portion of the Penn Treebank II (sections 02-21) that is covered by the mapping. Performance was evaluated using two test sets: section 23 of the Penn Treebank II, which represents the same genre as the training data; and the PropBank-annotated portion of the Brown corpus, which represents a very different genre.

3.7.3 Results

To compare the performance of these five systems on PropBank role labeling, we performed an evaluation where the probability assigned to each PropBank role was calculated by summing over the probabilities of all labels that use that PropBank role:

$$P(ArgN) = P(ArgN_{\text{agent}}) + P(ArgN_{\text{patient}}) + P(ArgN_{\text{destination}}) + ... \qquad (3.3)$$

Table 3.5 shows the performance of each system on the WSJ corpus section 23; and Table 3.6 shows the performance of each system on the Brown corpus. As expected, we see a significant drop in performance when we move from the baseline system (`PropBank`) to the system that uses subdivided labels without any subset features. This drop results from the sparse data problem – the model is no longer able to pool training data from different subdivisions of each PropBank argument. However, when we add the `PropBankSubset` features, the performance improves over the baseline, reflecting the fact that the model is able to make good use of some of the simple features. Similarly, the `VerbNetSubset` features yield an improvement over the baseline performance. Finally, including both the `PropBankSubset` features and the `VerbNetSubset` features yields an additional improvement.

System	**Precision**	**Recall**	F_1	
`PropBank` (baseline)	78.1	72.8	75.4	
`Simple`	72.4	68.1	70.2	$\star$
`Simple+PropBank`	78.7	73.6	76.1	$\star$
`Simple+VerbNet`	78.9	73.1	75.9	
`Simple+PropBank+VerbNet`	79.1	73.7	76.3	$\star$

Table 3.5: **Results from Experiment 3.7.3 (WSJ Corpus)**. Performance with fully subdivided labels, using different feature groups, tested using the *WSJ corpus (section 23)*. This represents performance on the same genre as the training corpus.

$\star$: score is significantly different from the baseline score.

System	**Precision**	**Recall**	F_1	
`PropBank` (baseline)	68.1	59.1	63.3	
`Simple`	63.6	54.8	58.9	$\star$
`Simple+PropBank`	68.8	60.2	64.2	$\star$
`Simple+VerbNet`	69.2	59.7	64.1	$\star$
`Simple+PropBank+VerbNet`	69.3	60.3	64.5	$\star$

Table 3.6: **Results from Experiment 3.7.3 (Brown Corpus)**. Performance with fully subdivided labels, using different feature groups, tested using the *PropBank-annotated portion of the Brown corpus*. This represents performance on a different genre from the training corpus.

$\star$: score is significantly different from the baseline score.

3.7.4 Discussion

As we have seen in this section, when faced with a task whose desired output labels may not form coherent classes in the feature space, it can be beneficial to subdivide those labels into smaller, more coherent labels. In particular, in cases like PropBank, where one label may be conflating multiple related but separate sub-problems, we should consider explicitly subdividing the label to capture those sub-problems.

When this subdivision is done naively, it comes with a high cost: the model for each subdivided label can be trained using only a subset of the training data that was used for the original label. However, we can overcome this problem by making use of "subset features," which allow the model to learn generalizations over separate but related labels. More generally, whenever we are working with a label space that has structure, we may want to consider defining features that reflect that structure, by grouping together related sets of labels.

3.7.5 Future Work

In this section, we explored the use of subset features to combine a very coarse grained label set (PropBank) with a much more fine-grained label set (VerbNet). However, it might also be beneficial to make use of intermediate levels of granularity, such as the "grouped" label set used in the experiments described in Section 3.6. As future work, I plan to explore the effectiveness of using subset features that include such intermediate levels of granularity.

The technique presented here, of taking the intersection of two labelings, and using subset features to allow the model to make generalizations, may also extend well to other domains where there is more than one plausible set of categories, which carry slightly different types of information. For example, this technique could be used to combine two word sense disambiguation label sets that make different distinctions among senses. The most likely word sense tag for either of the original label sets could then be calculated by selecting the tag with the highest marginal probability for that label set.

3.8 VerbNet Argument Classification

Because VerbNet role labels are defined in a verb-independent manner, they may prove more useful for tasks such as inferencing and question answering than PropBank numbered role labels. For example, if a question answering system is attempting to answer a question such as "how much did the price rise," then knowing that a verb's argument fills the "extent" role will be significantly more useful than knowing that it fills the "Arg2" role. However, since VerbNet uses 28 role labels for core arguments, compared to PropBank's 5, the task of classifying arguments by VerbNet role label is significantly more difficult than the task of classifying arguments by PropBank role labels. Thus, the benefit of the extra information provided by VerbNet role labels must be weighted against the expected decrease in predictive performance.

In addition to being a useful task in its own right, the VerbNet argument classification is useful for exploring the hypothesis that mapping a task to a coherent output space can improve machine learning performance. To that end, I built two VerbNet argument classification systems. The first system consists of two steps: first, the argument's PropBank label is predicted using a classifier trained on the original PropBank corpus; and then that PropBank label is mapped to a VerbNet label using the SemLink mapping. In order to ensure an equitable comparison, this mapping from PropBank argument labels to VerbNet argument labels is done using the gold-standard mapping that was used to generate training data for the second system. The second system directly predicts the argument's VerbNet role label, using a classifier trained on a version of the PropBank corpus where all the argument labels have been mapped to VerbNet labels. Both systems make use of partially subdivided labels, and include simple features and both subset feature groups (`VerbNetSubset`

System	**Precision**	**Recall**	F_1
Predict PropBank & Map	64.04	76.02	69.52
Predict VerbNet Directly	67.15	76.88	71.68 $\star$

Table 3.7: **Results from Experiment 3.8**. Performance of both VerbNet argument classification systems. The first system is a two-stage pipeline that first predicts the PropBank role; and then maps it to VerbNet. The second system directly predicts the VerbNet role. Both systems are tested using the *WSJ corpus (section 23).*

$\star$: score is significantly different from the predict&map score.

and `PropBankSubset`). In the first system, the probability of a PropBank argument label is taken to be the sum of the probabilities of all subdivided labels that use that PropBank label. And similarly, in the second system, the probability of a VerbNet label is the sum of the probabilities of all subdivided labels that use that VerbNet label.

Because we hypothesize that VerbNet role labels are more coherent than PropBank role labels, we expect this second system to show higher performance. This prediction is confirmed by Table 3.7, which compares the results for both systems. Note that these results are calculated with respect to VerbNet role labels, and are not directly comparable to the results for PropBank role labels.

3.9 Alternative Mapping Based on Frame Files

We have shown that the SemLink mapping can be used to improve SRL performance, by providing machine learning models with a more coherent label set. However, it is worth exploring whether other mappings would yield the same benefit.

I therefore created a second mapping by hand, based on the “frames files” that are included with the PropBank corpus. Each frames file provides textual descriptions for set of roles that a single verb can take. These descriptions are intended mainly for human consumption, and are not standardized in any way. For example, the role descriptions for the verb “eat” are “consumer, eater” (Arg0) and “meal” (Arg1).

Since Arg0 and Arg1 are already fairly consistent across verbs, I focused on

Destination
...end...
...new state...
...result...
...group...
...dest...
...destination...
...-to...
...given to...
...buyer...

Source
...substance...
...start...
...begin...
...source...
...-from...
...giver...
...had it before...
...taken from...
...medium...
...seller...

Amount
amount...
money...
...price...
...count...
value...

Instrument
...instrument...

Extent
...ext...
...extent...

Attribute
...attribute...
...attributive...

Beneficiary
...benefactive...
...beneficiary...

Table 3.8: **Argument Mapping Based on Frames Files**. Each table lists the set of patterns that were used to create a single mapped label. I.e., if a role's description matched any pattern in one of these tables, then it was mapped to that table's label.

creating a mapping for the remaining arguments (Arg2-5). I began by creating a list of all 1,193 unique descriptions used for roles Arg2-Arg5 (after normalizing case and discarding parentheticals). After examining this list, I incrementally created a set of patterns that merged role descriptions that described similar roles; and that did not merge any two roles from a single verb. In order to focus my search, I sorted the list of role descriptions by their frequency in both the lexicon (number of verbs) and the annotated corpus (number of verb occurrences). This set of patterns, which is shown in Table 3.8, took approximately 5 hours to create (including the time to create and sort the lists of role labels). It should be noted that this set of patterns only covers about 65% of the Arg2-5 occurrences in the PropBank corpus, and about 52% of the Arg2-Arg5 descriptions in the lexicon; see Table 3.9 for a breakdown of coverage by merged role.

The mapping that I created first checks these patterns, and if any pattern matches, then it assigns the corresponding label to the argument. Otherwise, it retains the original PropBank role label (such as Arg2) as the label. For example, the description for Arg2 of the verb "channel" is "hands"; since this does not match any of the argument mapping patterns, its label would simply be mapped to Arg2.

Argument	Coverage (corpus)	Coverage (lexicon)
destination	24.22%	11.28%
source	11.31%	10.70%
attribute	10.30%	6.54%
ext	6.82%	4.32%
amount	5.38%	2.18%
beneficiary	2.92%	5.54%
instrument	2.52%	10.20%
location	0.87%	1.25%
Total Coverage	64.35%	52.00%

Table 3.9: **Coverage of Argument Mapping Based on Frames Files**.

System	**Precision**	**Recall**	F_1	
`PropBank` (baseline)	78.1	72.8	75.4	
`Simple`	77.6	72.1	74.7	⋆
`Simple+PropBank`	78.5	73.3	75.8	
`Simple+Mapped`	78.7	73.2	75.9	
`Simple+PropBank+Mapped`	78.8	73.4	76.0	⋆

Table 3.10: **Results from Experiment 3.9 (WSJ Corpus)**. Performance with fully subdivided labels, using different feature groups, tested using the *WSJ corpus (section 23)*. This represents performance on the same genre as the training corpus.
⋆: score is significantly different from the baseline score.

In order to test the effectiveness of this new mapping, I repeated the experiments described in Section 3.7.3 using this new mapping. The results are given in Tables 3.10 and 3.11. The decrease in performance when moving from `PropBank` to `Simple` is significantly smaller than it was for experiment 3.7.3. This reflects the fact that we are fragmenting the set of role labels significantly less (8-way rather than 26-way); and that the mapping did not apply to arguments Arg0 and Arg1. Once we add in the subset features, the performance improves over the baseline; but the improvement is smaller than it was for the SemLink mapping.

System	**Precision**	**Recall**	F_1
PropBank (baseline)	68.1	59.1	63.3
Simple	63.6	54.8	58.9 $\star$
Simple+PropBank	68.8	60.2	64.2 $\star$
Simple+Mapped	69.2	59.7	64.1 $\star$
Simple+PropBank+Mapped	69.3	60.3	64.5 $\star$

Table 3.11: **Results from Experiment 3.9 (Brown Corpus)**. Performance with fully subdivided labels, using different feature groups, tested using the *PropBank-annotated portion of the Brown corpus*. This represents performance on a different genre from the training corpus.

$\star$: score is significantly different from the baseline score.

3.10 Conclusion

Using SRL argument classification as an example, we have shown that transforming the output encoding for a simple classification task can improve performance. In particular, by replacing output labels that conflate multiple distinctions with subdivided labels that tease out those distinctions, we allow the model to more accurately capture the sets of features that should predict to a given label. Furthermore, by making use of subset features, we also allow the model to make generalizations over groups of subdivided labels, where appropriate.

Chapter 4

Using Search to Optimize Output Encodings for Sequence Prediction

As we have seen, output encoding transformations can improve performance for simple classification problems such as SRL argument identification. In this chapter, we show that output transformations can also be beneficial for structured prediction problems. In particular, we examine three sequence prediction tasks; and show that by choosing appropriate transformations to the label sequences, we can "rephrase" each task in such a way that machine learning methods are better able to model them.

Section 4.1 gives brief descriptions of the the three structured prediction tasks that are examined in this chapter. Section 4.2 describes an example of a hand-crafted encoding transformation that improves NP chunking performance. Section 4.3 introduces the idea of using search techniques to automatically find output encoding transformations that improve system performance. Section 4.4 describes how Finite State Transducers (FSTs) can be used to define the search space for this process, and Section 4.5 presents a set of FST modification operations that can be used to move through this search space. Section 4.6 describes a simple hill-climbing algorithm that can be used to explore the search space, and Section 4.7 presents an optimization technique that can be used to significantly increase the search speed. Section 4.8 describes experimental results of applying the hill-climbing algorithm to the tasks of part of speech tagging, noun phrase chunking, and bio-entity recognition, using linear chain Conditional Random Fields (CRFs) as the underlying model. Finally, Section 4.9 presents a set of experiments exploring the effectiveness of the hill-climbing algorithm when used with simple Hidden Markov Models (HMMs) rather than CRFs.

4.1 Sequence Prediction Tasks

4.1.1 Part of Speech Tagging

Part of speech tagging is the task of labeling each word in a sentence with a tag describing its lexical category. These tags are chosen from a fixed set, and typically encode information about the syntactic and morphological behavior of the word. For this task, I used the Penn Treebank II, which uses tags such as DT (determiner), NNP (proper noun), and VBN (past participle verb). In the following example sentence, each word is labeled with its part of speech tag:

> *Chancellor*$_{NNP}$ *of*$_{IN}$ *the*$_{DT}$ *Exchequer*$_{NNP}$ *Nigel*$_{NNP}$ *Lawson*$_{NNP}$ *'s*$_{POS}$ *restated*$_{VBN}$
> *commitment*$_{NN}$ *to*$_{TO}$ *a*$_{DT}$ *firm*$_{NN}$ *monetary*$_{JJ}$ *policy*$_{NN}$ *has*$_{VBZ}$ *helped*$_{VBN}$
> *to*$_{TO}$ *prevent*$_{VB}$ *a*$_{DT}$ *freefall*$_{NN}$

4.1.2 Noun Phrase Chunking

Noun phrase chunking is the task of finding the non-overlapping sub-sequences of a sentence that form the non-recursive portions of noun phrases. For this task, I used the “RM95” corpus described in Ramshaw and Marcus 1995 (Ramshaw and Marcus, 1995), which was derived automatically from the Penn Treebank. In the following example sentence, noun phrase chunks are underlined and marked by brackets:

> *[Confidence] in [the pound] is widely expected to take [another sharp dive] if [trade figures] for [September], due for [release] [tomorrow], fail to show [a substantial improvement] from [July and August] ['s near-record deficits].*

The noun phrase task can be encoded as a sequence prediction task by marking each word in the sentence with a label indicating whether it is part of a chunk. The RM95 corpus uses the following set of labels to encode the chunk structure of a sentence:

- `I`: This word is *inside* (i.e., part of) a chunk.
- `O`: This word is *outside* (i.e., not part of) a chunk.
- `B`: This word is at the *boundary* of two adjacent chunks, and begins a new chunk.

The tag sequence for the example sentence given above is:

> *Confidence*$_I$ *in*$_O$ *the*$_I$ *pound*$_I$ *is*$_O$ *widely*$_O$ *expected*$_O$ *to*$_O$ *take*$_O$ *another*$_I$ *sharp*$_I$ *dive*$_I$ *if*$_O$ *trade*$_I$ *figures*$_I$ *for*$_O$ *September*$_I$, *due*$_O$ *for*$_O$ *release*$_I$ *tomorrow*$_B$, *fail*$_O$ *to*$_O$ *show*$_O$ *a*$_I$ *substantial*$_I$ *improvement*$_I$ *from*$_O$ *July*$_I$ *and*$_I$ *August*$_I$ *'s*$_B$ *near-record*$_I$ *deficits*$_I$ *.*$_O$

4.1.3 Bio-Entity Recognition

Bio-Entity Recognition is the task of identifying and classifying technical terms in the domain of molecular biology, such as genes, proteins, and cell lines. For this task, I used the JNLPBA bio-entity recognition corpus, which was derived from the Genia corpus and distributed for the JNLPBA shared task in bio-entity recognition (Kim et al., 2004). This corpus identifies and classifies all terms referring to a protein, DNA sequence, RNA sequence, cell line, and cell type. In the following sentence, these terms have been underlined and marked with brackets:

> *We have shown that [interleukin-1]*$_{\textit{protein}}$ *([IL-1]*$_{\textit{protein}}$*) and [IL-2]*$_{\textit{protein}}$ *control [IL-2 receptor alpha (IL-2R alpha) gene]*$_{\textit{DNA}}$ *transcription in [CD4-CD8-murine T lymphocyte precursors]*$_{\textit{cell-line}}$*.*

The bio-entity recognition corpus encodes the location and types of technical terms in a sentence using a sequence of tags, similar to those used by RM95:

- `B-`***type***: This word is at the *beginning* of a chunk of type *type*.
- `I-`***type***: This word is *inside* (i.e., a continuation of) a chunk of type *type*.
- `O`: This word is *outside* (i.e., not part of) a chunk.

Where *type* may be "protein," "DNA," "RNA," "cell-line," or "cell-type." The tag sequence for the example sentence given above is:

> *We*$_{O}$ *have*$_{O}$ *shown*$_{O}$ *that*$_{O}$ *interleukin-1*$_{\textit{B-protein}}$ *(*$_{O}$ *IL-1*$_{\textit{B-protein}}$ *)*$_{O}$ *and*$_{O}$ *IL-2*$_{\textit{B-protein}}$ *control*$_{O}$ *IL-2*$_{\textit{B-DNA}}$ *receptor*$_{\textit{I-DNA}}$ *alpha*$_{\textit{I-DNA}}$ *(*$_{\textit{I-DNA}}$ *IL-2R*$_{\textit{I-DNA}}$ *alpha*$_{\textit{I-DNA}}$ *)*$_{\textit{I-DNA}}$ *gene*$_{\textit{I-DNA}}$ *transcription*$_{O}$ *in*$_{O}$ *CD4-CD8-murine*$_{\textit{B-cell-line}}$ *T*$_{\textit{I-cell-line}}$ *lymphocyte*$_{\textit{I-cell-line}}$ *precursors*$_{\textit{I-cell-line}}$ *.*$_{O}$

4.2 Improving Chunking Performance with a Hand-Crafted Encoding Transformation

In Chapter 3, we saw an example of a case where the labels used by a classification model were grouping together related but distinct sub-classes. This made it difficult for models to capture the overall classes accurately, since they had no way of capturing the distinctions between subclasses. We were able to overcome this problem by splitting the labels to reflect the sub-classes, thereby allowing the model to learn their distinct characteristic behaviors.

This same basic principle can be applied to the task of NP chunking. In particular, there are a variety of ways that the labels used to encode NP chunks can be split,

```
for each NP chunk $c$:
    if any word in $c$ has tag NNP (proper noun):
        append -proper to each tag in $c$
    else if any word in $c$ has tag CC (number) and no word has tag NN:
        append -numeric to each tag in $c$
    else if any word in $c$ has tag NN:
        append -other to each tag in $c$
```

Figure 4.1: **Hand-Crafted Transformation for Chunk Encodings**. This transformation splits each of the original tags (**I**, **O**, and **B**) based on whether they occur in a proper NP chunk, a numeric NP chunk, or any other type of NP chunk.

to reflect different specialized noun phrase chunk types. After a brief examination of the RM95 chunking corpus, I selected two groups of noun phrase chunks that appeared to have unique behavior: proper nouns, and numeric expressions.

4.2.1 Hand-Crafted Encoding Transformation for NP Chunking

In order to test the hypothesis that we could exploit this unique behavior to improve NP chunking performance, I transformed the corpus' IOB1 encoding to a new encoding that encoded the difference between proper NP chunks, numeric NP chunks, and all other NP chunks, by splitting each of the original IOB1 tags into three subclasses. This transformation is shown in Figure 4.1, and the resulting tagset is shown in Figure 4.2.

4.2.2 Evaluating the Hand-Crafted Encoding Transformation

I then compared the performance of a model trained using this new tag set to the performance of a model using the original IOB1 tag set. Both models were linear chain Conditional Random Fields (CRFs), and were trained using the Mallet toolkit

Tag	Description
B-proper	Boundary word between two chunks, starting a new proper noun chunk.
B-numeric	Boundary word between two chunks, starting a new numeric noun chunk.
B-other	Boundary word between two chunks, starting any other new chunk.
I-proper	Non-boundary chunk word inside a proper noun chunk
I-numeric	Non-boundary chunk word inside a numeric noun chunk
I-other	Non-boundary chunk word inside any other noun chunk
O	Non-chunk word

Figure 4.2: **Hand-Crafted Chunk Encodings Tags**. The tags generated by the transformation described in Figure 4.1.

(McCallum, 2002). The feature set used by both models, which was based on Sha & Pereira (2003) (Sha and Pereira, 2003), is shown in Figure 4.3.

The results of this comparison are shown in the first two lines of Table 4.1. As we can see, the hand-crafted encoding does not perform any better than the original model. However, this result isn't entirely unexpected – as we saw for argument classification, simple label splitting has both beneficial and detrimental effects. In particular, although splitting labels increases the model's ability to capture differences between different NP chunk types, this benefit is counterbalanced by a decrease in the amount of training data that can be used to model each tag.

In the case of argument classification, we overcame this problem by making use of "subset features," which were essentially shared by multiple labels. We can use a similar approach for sequence prediction tasks such as NP chunking. Like the conditional exponential models used by MaxEnt, linear chain CRFs define features as functions of both the input and the output. We can therefore define features that will fire for a given subset of output labels. These features allow the model to generalize over the distinctions that are made by subdividing the output labels.

To test the effectiveness of this approach, I built a third model for the NP chunking task, based on the hand-crafted encoding with subdivided labels for proper NP chunks and numeric NP chunks. I provided this model with features based on each of the individual subdivided labels, along with subset features that ignored these

Feature	Description
y_i	The current output tag.
y_i, w_{i+n}	A tuple of the current output tag and the $i + n$th word, $-2 \leq n \leq 2$.
y_i, w_i, w_{i-1}	A tuple of the current output tag, the current word, and the previous word.
y_i, w_i, w_{i+1}	A tuple of the current output tag, the current word, and the next word.
y_i, t_{i+n}	A tuple of the current output tag and the part of speech tag of the $i + n$th word, $-2 \leq n \leq 2$.
y_i, t_{i+n}, t_{i+n+1}	A tuple of the current output tag and the two consecutive part of speech tags starting at word $i + n$, $-2 \leq n \leq 1$.
$y_i, t_{i+n-1}, t_{i+n}, t_{i+n+1}$	A tuple of the current output tag, and three consecutive part of speech tags centered on word $i + n$, $-1 \leq n \leq 1$.

Figure 4.3: **Feature Set for the CRF NP Chunker**. y_i is the i^{th} output tag; w_i is the i^{th} word; and t_i is the part-of-speech tag for the i^{th} word.

System	**Precision**	**Recall**	F_1
Baseline Encoding (IOB1)	93.56	93.82	93.69
Hand-Crafted Encoding	93.64	93.71	93.67
Hand-Crafted Encoding w/ Subset Features	94.18	94.23	94.20 $\star$

Table 4.1: **Hand-Crafted Chunk Transformation Results**.
$\star$: score is significantly different from the baseline score.

subdivisions. The performance of this third model is compared to the performance of the first two models in Table 4.1. This new model is able to directly model the idiosyncrasies of proper NP chunks and numeric NP chunks, without suffering the sparse data problems that decreased the performance of the second model.

4.2.3 Modelling Long-Distance Output Dependencies

The transformation of the encoding used to represent the NP chunking problem can affect the model's performance in two ways. First, it can allow the model to more accurately model the local sub-problem of scoring individual labels, by replacing a less coherent label with a set of more coherent labels. This local effect is entirely responsible for the improvements seen in Chapter 3 for SRL argument classification; and is responsible for much of the improvement seen in the hand-crafted encoding discussed in this section.

But there is also a second potential benefit to transforming the output encoding that applies only to structured prediction (and not to simple classification): it can increase the set of long-distance dependencies between output tags that the model is able to learn. In dynamic programming models such as HMMs, MEMMs, and CRFs, the score for each label may only depend on the values of its immediate neighbor labels. By subdividing the labels, we increase the information content of those labels, thereby allowing more information to be shared between distant sub-problems.

We can see one example of this advantage by examining how the different models perform for conjunctions such as "and." Conjunctions are often considered part of a proper noun; but they usually divide common nouns. However, it's not always apparent from the immediate context of the conjunction whether it is part of a common noun or a proper noun. For example, consider the NP chunk "Republican presidential and senatorial candidates." By using separate labels for proper and common nouns, the model can capture the fact that this chunk is a proper noun (which is only reflected in the first word of the chunk); and then make use of that

System	**Accuracy**
Baseline Encoding (IOB1)	91.13
Hand-Crafted Encoding w/ Subset Features	91.63

Table 4.2: **Conjunction Modeling Accuracy**. The accuracy of each model at predicting whether a conjunction will be part of a noun phrase chunk (I or B) or not part of a chunk (O).

fact once it examines the conjunction "and."

In order to evaluate the magnitude of this effect for the hand-crafted chunk encoding, I calculated the accuracy with which each model predicted whether conjunctions were part of a noun phrase or not. The results, listed in Table 4.2, show that the hand-crafted encoding does improve performance on this task slightly. However, for this task and this encoding, I believe that most of the benefit in chunking performance comes from improving the model's ability to accurately model individual sub-problems; and not from the extended set of long-distance dependencies that are available to the model.

4.3 Using Search to Find Good Encoding Transformations

As we've seen in Section 4.2, encoding transformations can improve the performance of sequence prediction systems, both by improving their ability to model local sub-problems and by affecting the set of long-distance dependencies between pieces of output structure that they can learn. In the remainder of this Chapter, we will explore the use of search techniques to automatically find output encoding transformations that improve system performance. In order to apply search techniques, we must first define:

1. The search space. I.e., concrete representations for the set of encoding transformations we will consider.

2. A set of operations for moving through that search space.
3. An evaluation metric.

In Section 4.4, we will define the search space by using Finite State Transducers (FSTs) to represent individual encoding transformations; and in Section 4.5, we will define the set of operations for moving through the search space as modification operations on FSTs. For the evaluation metric, we simply train and test a model based on a given FST's transformed encoding on held-out data. In particular, to score a given FST, we first use it to transform the training corpus, and train a CRF on that transformed corpus. We then apply this CRF to a heldout corpus, and use an inverted FST to transform its output back to the original encoding. Finally, we evaluate the system's results on the heldout corpus, using either accuracy (for part of speech tagging) or F-measure (for NP chunking and bio-entity detection).

4.4 Representing Sequence Encodings with FST

As was discussed in Chapter 1, output encodings can be defined using reversible transformations with respect to a chosen canonical encoding. Finite State Transducers (FSTs) provide a natural formalism for representing output transformations for sequence prediction tasks. FSTs are powerful enough to capture a wide variety of encoding transformations, including transformations that add history information, such as moving from a first-order to a second-order model; and transformations that modify the set of classes used to encode different output pieces, such as moving from the `IOB1` chunk encoding to `IOB2` or IOBES. FSTs are efficient, so they add very little overhead. Finally, there exist standard algorithms for inverting and determinizing FSTs. [1]

[1] Note that we are not attempting to learn a transducer that generates the output values from input values, as is done in e.g. (Oncina et al., 1993) and (Stolcke and Omohundro, 1993). Rather, we we are interested in finding a transducer from one output encoding to another output encoding that will be more amenable to learning by the underlying learning algorithm.

We will use the NP chunking task as a running example to illustrate how FSTs can be used to transform encodings. We take `IOB1` as the canonical encoding, since it is the encoding used by the RM95 corpus.[2]

Given an encoding's FST, we can encode an output structure by first taking the canonical encoding for that structure; and then applying the FST to that canonical encoding. To decode a tag sequence, we first apply the inverted FST to the tag sequence, to get a canonically-encoded tag sequence; and then decode that tag sequence into an output structure, using the canonical encoding. An example of this encoding and decoding procedure is illustrated in Figure 4.4. Figure 4.5 shows the FSTs that can be used to represent five common NP chunking encodings.

4.4.1 FST Model & Notation

Without loss of generality, I will assume an FST model with the following properties:

- Each arc maps a single input symbol to an output symbol string. As a result, there are no epsilon-input arcs; but epsilon-output arcs are allowed.

- There is a single initial state.

- Each final state has a (possibly empty) 'finalization string,' which will be generated if the FST terminates at that state.

The variables S, Q, and P will be used for states. The variables x, y, and z will be used for symbols. The variables α, β, and γ will be used for (possibly empty) symbol strings. Arcs will be written as $\langle S \rightarrow Q[\alpha : \beta] \rangle$, indicating an arc from state S to state Q with input string α and output string β.

[2] Since any of the other commonly used tag-based encodings can be transformed to `IOB1` by an FST, and since FSTs are closed under composition, we are guaranteed that this choice of canonical encoding does not prevent us from expressing any encodings that a different canonical encoding, such as `IOB2`, would allow.

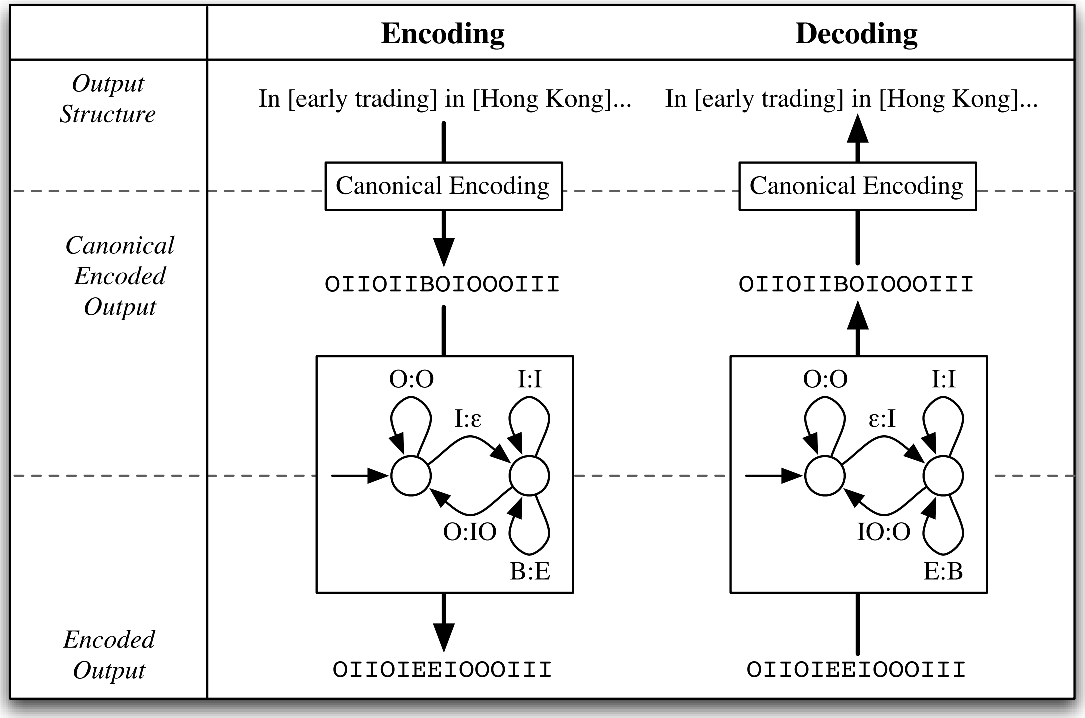


Figure 4.4: **Encoding Chunk Sequences with FSTs**. Output encodings are represented using FSTs. On the left, the FST for the IOE1 encoding is used to encode a chunk sequence, by first generating the canonical (IOB1) encoding, and then translating that encoding with the FST. On the right, that IOE1 encoding is decoded back to a chunk structure by first applying the inverted IOE1 FST; and then interpreting the resulting string using IOB1.

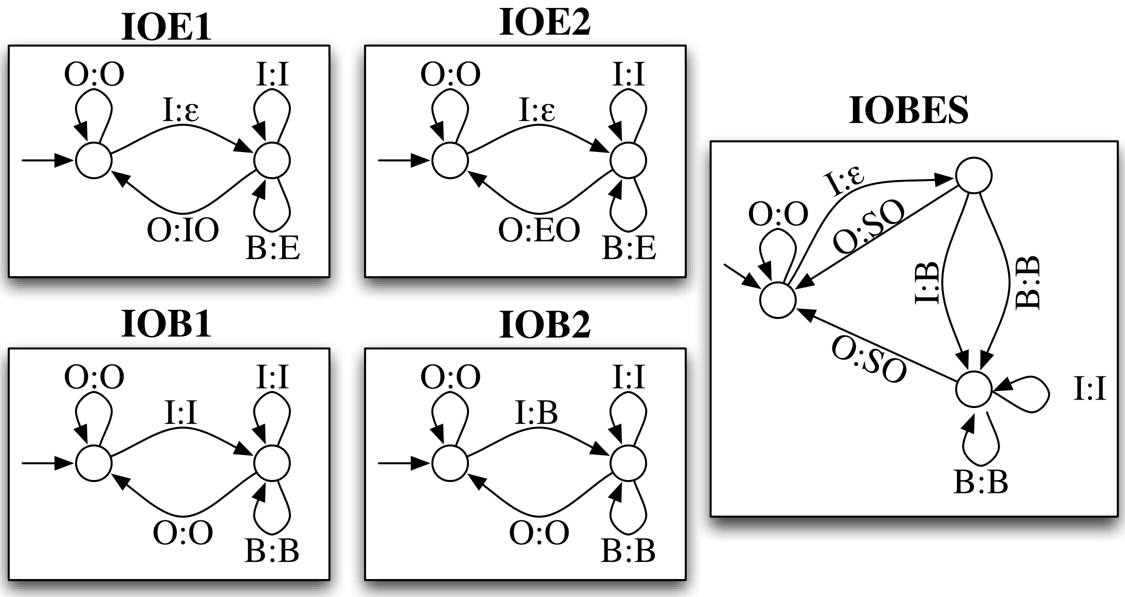


Figure 4.5: **FSTs for Five Common Chunk Encodings**. Each transducer takes an `IOB1`-encoded string for a given output value, and generates the corresponding string for the same output value, using a new encoding. Note that the `IOB1` FST is simply an identity transducer; and note that the transducers that make use of the `E` tag must use ϵ-output edges to delay the decision of which tag should be used until enough information is available.

4.4.2 Necessary Properties for Representing Encodings with FSTs

In order for an FST to be used to transform output values, it must have the following three properties:

1. The FST's inverse should be deterministic.[3] Otherwise, we will be unable to convert the model's (transformed) output into an un-transformed output value.

2. The FST should recognize exactly the set of valid output values.

   – If it does not recognize some valid output value, then it won't be able to transform that value.
   – If it recognizes some invalid output value, then there exists a transformed output value that would map back to an invalid output value.

3. The FST should not modify the length of the output sequence. Otherwise, it will not be possible to align the output values with input values when running the model.

In addition, it seems desirable for the FST to have the following two properties:

4. The FST should be deterministic. Otherwise, a single training example's output could be encoded in multiple ways, which would make training the individual base decision classifiers difficult.

5. The FST should generate every output string. Otherwise, there would be some possible system output that we are unable to map back to an un-transformed output.

Unfortunately, these two properties, when taken together with the first three, are problematic. To see why, assume an FST with an output alphabet of size k. Property

[3] Or at least determinizable.

(5) requires that all possible output strings be generated, and property (1) requires that no string is generated for two input strings, so the number of strings generated for an input of length n must be exactly k^n. But the number of possible chunkings for an input of length n is $3^n - 3^{n-1} - (n-2)3^{n-2}$; and there is no integer k such that $k^n = 3^n - 3^{n-1} - (n-2)3^{n-2}$.[4]

We must therefore relax at least one of these two properties. Relaxing property 4 (deterministic FSTs) will make training harder; and relaxing property 5 (complete FSTs) will make testing harder. In the experiments presented here, we chose to relax the second property.

Inverting the Transformation

Recall that the motivation behind property 5 is that we need a way to map *any* output generated by the machine learning system back to an un-transformed output value.

As an alternative to requiring that the FST generate every output string, we can define an extended inversion function, that includes the inverted FST, but also generates output values for transformed values that are not generated by the FST. In particular, in cases where the transformed value is not generated by the FST, we can assume that one or more of the transformed tags was chosen incorrectly; and make the minimal set of changes to those tags that results in a string that *is* generated by the FST. Thus, we can compute the optimal un-transformed output value corresponding to each transformed output using the following procedure:

1. Invert the original FST. I.e., replace each arc $\langle S \rightarrow Q[\alpha : \beta]\rangle$ with an arc $\langle S \rightarrow Q[\beta : \alpha]\rangle$.

2. Normalize the FST such that each arc has exactly one input symbol.

[4] To see why the number of possible chunkings is $3^n - 3^{n-1} - (n-2)3^{n-2}$, consider the IOB1 encoding: it generates all chunkings, and is valid for any of the 3^n strings except those that start with `B` (of which there are 3^{n-1}) and those that include the sequence `OB` (of which there are $(n-2)3^{n-2}$).

3. Convert the FST to a weighted FST by assigning a weight of zero to all arcs. This weighted FST uses non-negative real-valued weights, and the weight of a path is the sum of the weights of all edges in that path.

4. For each arc $\langle S \rightarrow Q[x : \alpha]\rangle$, and each $y \neq x$, add a new arc $\langle S \rightarrow Q[y : \alpha]\rangle$ with a weight one.

5. Determinize the resulting FST, using a variant of the algorithm presented in (Mohri, 1997). This determinization algorithm will prune paths that have non-optimal weights. In cases where the determinization algorithm has not completed by the time it creates 10,000 states, the candidate FST is assumed to be non-determinizable, and the original FST is rejected as a candidate.

The resulting FST will accept all sequences of transformed tags, and will generate for each transformed tag the un-transformed output value that is generated with the fewest number of "repairs" made to the transformed tags.

4.4.3 FST Characteristics

In the introduction to this section, we defined three properties that an FST must have in order to represent an output encoding. Based on these properties, we can derive some additional characteristics that such an FST must have.

First, the FST may not contain any loops that could cause the input and output lengths to differ:

Theorem 4.1. *In an output encoding FST, for any cycle, the total length of the arcs' input strings must equal the total length of the arcs' output strings. (Unless the cycle is not on any path from the initial state to a final state – in which case it has no effect on the behavior of the transducer.)*

Proof. For any cyclic path c from state s to s, consider two paths from the initial state to a final state: p_1 passes through state s, but does include the cycle; and p_2 is the same path, but with one turn through the cycle. Let $in(p)$ be the input string recognized by a path p, and $out(p)$ be the output string generated by that path. By assumption, $|in(p)| = |out(p)|$ for any path from an initial node to a final node. Thus $|in(p_1)| = |out(p_1)|$ and $|in(p_2)| = |out(p_2)|$. But $|in(p_2\ )| = |in(p_1)| + |in(c)|$ and $|out(p_2)| = |out(p_1)| + |out(c)|$. Therefore $|in(c)| = |out(c)|$. □

Next, we can associate a unique number, the $\mathsf{output\ offset}$, with each state, which specifies the difference between the length of the input string consumed and the output string generated whenever we are at that state.

Theorem 4.2. *For each state S, there exists a unique integer* $\mathsf{output\ offset}(S)$*, such that on any path p from the initial state to S,* $|in(p)| - |out(p)| = \mathsf{output\ offset}(S)$

Proof. Let p_1 and p_2 be two paths from the initial state to S, and let p_3 be a path from S to a final state. Then by assumption:

$$|in(p_1)| + |in(p_3)| = |out(p_1)| + |out(p_3)|$$

$$|in(p_2)| + |in(p_3)| = |out(p_2)| + |out(p_3)|$$

Rearranging and substituting gives:

$$|in(p_1)| - |out(p_1)| = |in(p_2)| - |out(p_2)|$$

Therefore, $\mathsf{output\ offset}(S)$ is unique. □

4.4.4 FST Input Symbols

As we saw in Section 4.2, it can be useful to define encoding transformations that depend on the token sequence whose output is being encoded. In particular, we defined a transformation that only applied to a chunk's tags if one of the chunk's tokens was a proper noun.

But in a basic output encoding FST, the input symbols are just the set of tags used by the original (canonical) encoding. Using this set of input symbols, there is no way for an FST to express any dependence on the token sequence.

We will therefore make use of specialized input symbols, called "feature symbols," each of which combines a single tag from the original (canonical) encoding with zero or more "feature conditions."[5] Each feature condition either requires that a given token has a given property, or requires that it does not have a given property. For example, the feature condition $[stem = run]$ requires that a token's word stem be "run." Symbols with feature conditions are written using subscripts describing the feature conditions. For example, "$I_{[stem=run][pos \neq NN]}$" is an input symbol that will match any word whose canonical tag is I, whose word stem is "run," and whose part of speech tag is not "NN."

When using the FST to transform a token sequence's output tags, an input symbol is considered to match a tag from the original encoding if and only if the symbol's tag value is equal to that tag and all of its feature conditions are satisfied by the token corresponding to that tag.

When inverting the FST, to decode the new encoding back to the canonical output, we simply discard all feature conditions, since they should be redundant.

4.4.5 FST Output Symbols

When transforming an output encoding, it is important to avoid the sparse data problems that can arise when the original tags are split into more fine-grained tags. As we saw for both SRL argument classification (Chapter 3 and the hand-crafted chunking transform discussed in Section 4.2, the reduction in the amount of training data available for each tag can negate the benefits that come from being able to model distinctions between the different sub-tags. In both cases, we overcame this

[5] Note that in this context, "feature" refers to a property of an input token, and not a function from an input and an output value to a number.

Label	Feature Group								
	I_*	$I_{1,*}$	$I_{2,*}$	$I_{2,1_*}$	$I_{2,2_*}$	O_*	$O_{1,*}$	$O_{2,*}$	B_*
I_1	X			X					
$I_{2,1}$	X	X			X				
$I_{2,2}$	X	X				X			
O_1			X				X		
O_2			X					X	
B									X

Figure 4.6: **Example Feature Groups for Complex Output Symbols**. This table shows the set of feature groups that would be created for a transformed feature encoding that uses the following tag set: $\{I_1, I_{2,1}, I_{2,2}, O_1, O_2, B\}$. Each column is a single feature group, pairing every context function $g(x)$ with a test that the class label is one of the values marked with "X" for that column.

problem by supplying the learning algorithm with "subset features," which could be used to capture the behavior that is shared between different sub-tags of a given original tag.

In order to allow the output encodings generated by our FSTs to make use of subset features, we use output symbols that consist of a simple tag followed by zero or more subscripts, such as NP_2, I, or $B - DNA_{1,2,3}$. The FST treats each of these labels as atomic units, and considers two labels equal only if their tag and all their subscripts are equal. However, when training the CRF, we add a group of subset features corresponding to each tag prefix. For example, Figure 4.6 shows the set of feature groups that would be generated for a transformed feature encoding that uses the tag set $\{I_1, I_{2,1}, I_{2,2}, O_1, O_2, B\}$.

4.5 Sequence-Encoding FST Modifications

In order to perform search in the space of output-transforming FSTs, we must define a set of modification operations that generate a new FST from a previous FST. We will begin by defining three general operations which are sufficient to generate any sequence-encoding transformation that can be represented by an FST. However, the first two of these operations can be used to make very drastic changes to the transformation defined by an FST, and are therefore not very amenable to incremental search. Therefore, we define a number of more specific modification operations which instantiate particularly useful combinations of these three more general operations.

4.5.1 General Modification Operations

We begin by defining three general operations, state splitting and output relabeling, and feature specialization, which are sufficient to generate any sequence-encoding transformation that can be represented by an FST.

Output Relabeling

The output relabeling operation replaces the output strings on a state S's incoming and outgoing arcs with new output strings, with the following restrictions:

- The FST's inverse must remain deterministic. I.e., the output strings may not be modified in such a way that multiple input values will generate the same output value. This ensures that we can map the new encoding back to the canonical encoding.

- If the lengths of the output strings are changed, then there must be a unique (possibly negative) n such that the lengths of all incoming edges' output strings is increased by n; the lengths of all outgoing edges' output strings is decreased by n; and the length of the finalization string is decreased by n (if the state is a final state). This will ensure that $\mathsf{output\,offset}(S)$ remains unique for the state; the new value for $\mathsf{output\,offset}(S)$ will differ from the original value for $\mathsf{output\,offset}(S)$ by n.

State Splitting

The state splitting operation is used to introduce new structure to the sequence-encoding FST, by increasing the number of states it contains. This operation does not, by itself, make any change to the transduction defined by the FST; however, by adding new structure, it enables other modification operations to change the sequence-encoding transduction in new ways.

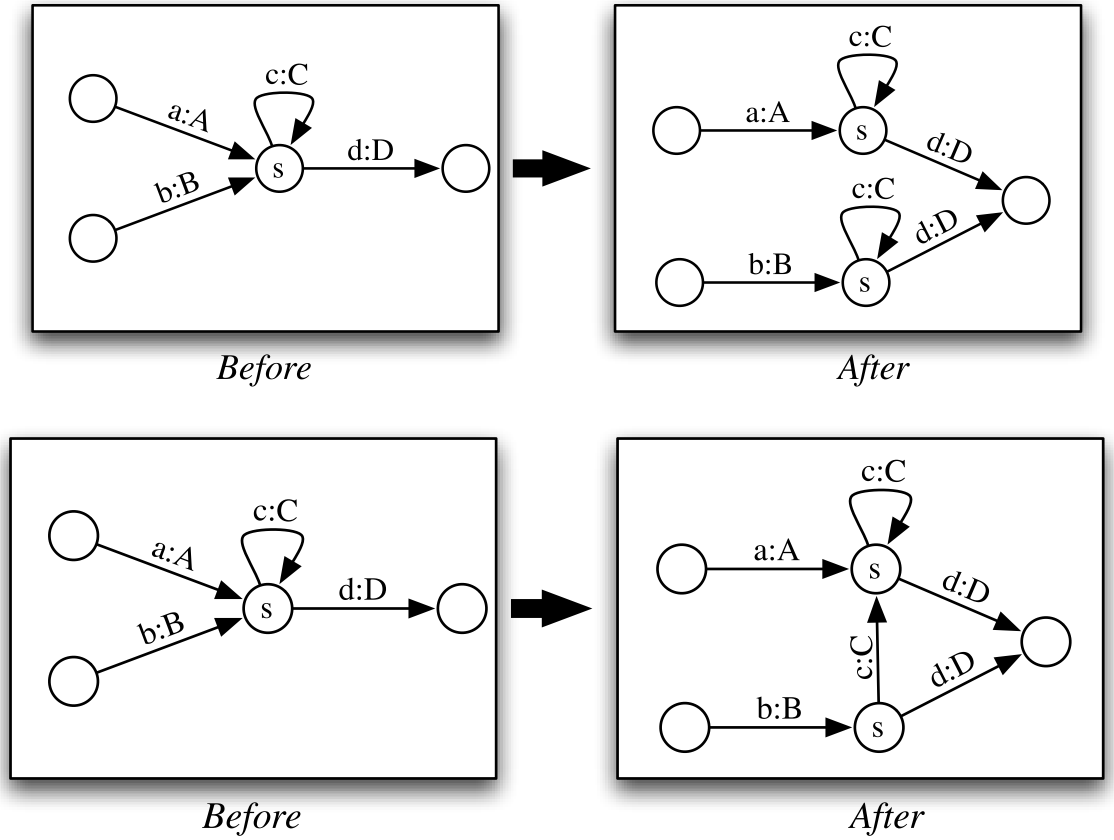
The state splitting operation replaces an existing state in the graph with two new equivalent states, and divides the incoming arcs to the original state between the two new states. It is parametrized by a state, a subset of that state's incoming arcs called the *redirected arc set*, and a subset of the state's loop arcs called the *copied loop set*. The state splitting operation makes the following changes to an FST:

1. A selected state S is duplicated. I.e., a new state, S' is created, with the same finalizing sequence as S; and for each outgoing arc from S, a corresponding arc is added to S'. In particular, for each arc $\langle S \rightarrow Q[\alpha : \beta]\rangle$, add a new arc $\langle S' \rightarrow Q[\alpha : \beta]\rangle$. Note that self-loop arcs from S (i.e., arcs where $Q = S$) will result in arcs from S' to S.

2. For each incoming arc in the redirected arc set, change the arc's destination from S to S'.

3. For each arc in the copied loop set, we added a corresponding arc from S' to S in step 1. Change this arc's destination from S to S' (turning it into a self-loop arc at S').

Figure 4.7 shows two examples of the state splitting operation.



Before *After*

Before *After*

Figure 4.7: **Two Examples of the State Splitting Operation.**

Feature Specialization

The feature specialization operation is used to introduce a dependency on a feature of the token sequence, by adding a feature condition to an arc's input symbol.[6] In particular, this operation replaces a single arc $\langle S \rightarrow Q[x : \beta]\rangle$ with two new arcs $\langle S \rightarrow Q[x_{[f=v]} : \beta]\rangle$ and $\langle S \rightarrow Q[x_{[f\neq v]} : \beta]\rangle$.

Like the state splitting operation, this operation does not directly modify the transformation defined by the FST; however, by adding new structure that depends on token features, it enables other modification operations to change the sequence-encoding transduction in new ways.

4.5.2 Search-Friendly Modifications

In order to support an incremental search strategy, the modification operations we define should make small incremental changes to the FSTs. Additionally, each modification operation should have some effect on the transformation defined by the FST. The selection of appropriate modification operations is important, since it will significantly impact the efficiency of the search process. In this section, I describe the set of FST modification operations that are used for the experiments described in this chapter. These operations were chosen based on intuitions about what modifications would support efficient hill-climbing search.

New Output Tag

The new output tag operation replaces an arc $\langle S \rightarrow Q[\alpha : \beta x\gamma]\rangle$ with an arc $\langle S \rightarrow Q[\alpha : \beta y\gamma]\rangle$, where y is a new output tag that is not used anywhere else in the transducer. When a single output tag appears on multiple arcs, this operation effectively splits that tag in two. For example, when applied to the transducer for the IOB2 encoding shown in Figure 4.5, this operation can be used to distinguish

[6] Recall that in Section 4.4.1, we stated that our FSTs would be kept in a normalized form where each arc has a single input symbol.

B tags that follow O tags from B tags that follow I or B tags – effectively creating a cross between IOB1 and IOB2 that has two separate tags for words that begin chunks, depending on whether they are also at the boundary between two adjacent chunks.

This operation is only applied if the output symbol x (or any subdivided symbol based on x) is used in at least one other location in the FST – otherwise, it would produce a simple global replacement of a single label, and would have no net effect on the overall system.

Specialize Output Tag

The specialize output tag operation is similar to the new output tag operation, but rather than replacing the output tag with a new tag, we subdivide the tag by adding a new subscript to the tag. In particular, this operation replaces an FST arc $\langle S \rightarrow Q[\alpha : \beta x \gamma]\rangle$ with an arc $\langle S \rightarrow Q[\alpha : \beta x_i \gamma]\rangle$. where i is the smallest integer such that x_i is not already used elsewhere in the transducer. As we discussed in Section 4.4.5, this will let the CRF model know that it should include both features for the subdivided tag x_i, and features that generalize over the subdivision by grouping all tags with prefix x.

This operation is only applied if the output symbol x (or any subdivided symbol based on x) is used in at least one other location in the FST – otherwise, it would simply subdivide a label into a single sub-group, which would have no effect.

This operation may be applied repeatedly, to further subdivide tags that have already been subdivided. For example, it could be used to replace an arc $\langle S \rightarrow Q[A : x_{1,2}]\rangle$ with an arc $\langle S \rightarrow Q[A : x_{1,2,1}]\rangle$.

Relabel Arc

The relabel arc operation replaces an arc $\langle S \rightarrow Q[\alpha : \beta]\rangle$ with an arc $\langle S \rightarrow Q[\alpha : \gamma]\rangle$, where γ is an output string that is composed of output symbols that are already used in the transducer. The new output string γ must be distinct from all other output strings and output string prefixes on outgoing edges from state S, to ensure that the FST remains invertible; and the length of the new output string must equal the length of the old output string, to ensure a consistent $\mathsf{output\,offset}(S)$ value. This operation can be used to recombine tags that have been split by previous operations to yield new combinations.

Loop Unrolling

The loop unrolling operation acts on a single self-loop arc e at a state S, and makes the following changes to the FST:

1. Create a new state S'.

2. For each outgoing arc $e_1 = \langle S \rightarrow Q[\alpha : \beta]\rangle \neq e$, add an arc $e_2 = \langle S' \rightarrow Q[\alpha : \beta]\rangle$. Note that if e_1 was a self-loop arc (i.e., $S = Q$), then e_2 will point from S' to S (e.g., the "c:C" arc in Figure 4.8).

3. Change the destination of loop arc e from S to S'.

Figure 4.8 shows the loop unrolling operation. The arc being unrolled is marked in bold. By itself, the loop unrolling operation just modifies the structure of the FST, but does not change the actual transduction performed by the FST. It is therefore always immediately followed by applying the new output tag operation or the specialize output tag operation to the loop arc e.

4.5.3 Output Delay

The output delay operation acts on a single state S, and requires that all of S's incoming edges have non-empty output strings. It makes the following changes:

- Strip the last output symbol off of each incoming edge's output string.

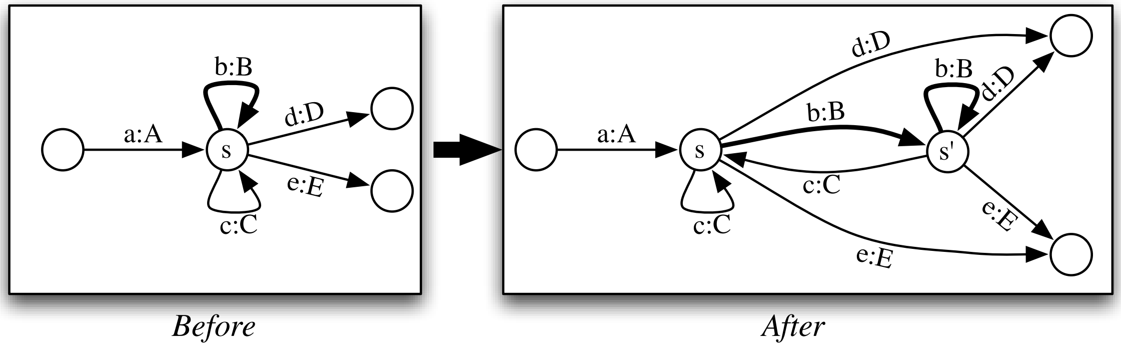


Figure 4.8: **Loop Unrolling**. The edge that is unrolled ("b:B") is marked in bold.

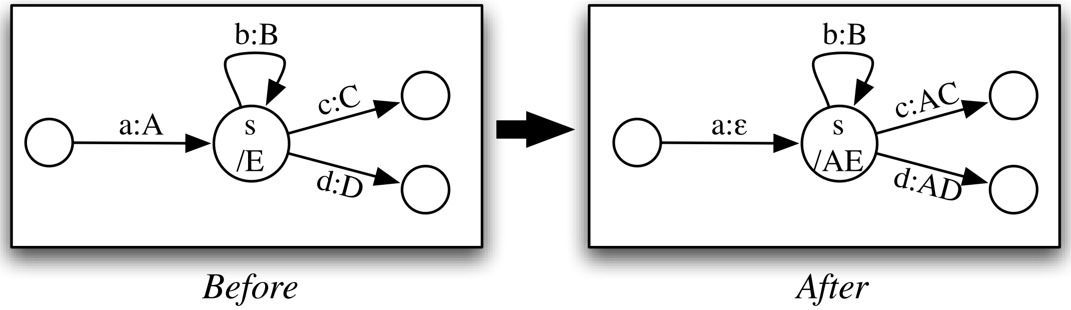


Figure 4.9: **Output Delay**.

- Add an output symbol to the beginning of each non-loop outgoing edge.
- Add an output symbol to the beginning of the finalization string of node S.

The output symbol that should be added to the non-loop outgoing edges and the finalization string can be either one of the stripped output symbols, or a new symbol. Figure 4.9 shows an example of the output delay operation. This operation allows the FST to shift the context in which decisions are made. For example, this operation can be used to transform the IOB encodings (where decisions about whether two adjacent words are at a boundary between chunks is made when looking at the beginning of the second chunk) into IOE encodings (where that decision is made when looking at the end of the first chunk).

Copy Tag Forward

The copy tag forward operation splits an existing state in two, directing all incoming edges that generate a designated output tag to one copy, and all remaining incoming edges to the other copy. The outgoing edges of these two states are then distinguished from one another, using either the specialize output tag operation or the new output tag operation.

This modification operation creates separate edges for different output histories, effectively increasing the "window size" of tags that pass through the state.

Copy State Forward

The copy state forward operation is similar to the copy tag forward operation; but rather than redirecting incoming edges based on what output tags they generate, it redirects incoming edges based on what state they originate from. This modification operation allows the FST to encode information about the history of states in the transformational FST as part of the model structure.

Copy Feature Forward

The copy feature forward operation is similar to the copy tag forward operation; but rather than redirecting incoming edges based on what output tags they generate, it redirects incoming edges based on a feature of the current input value. In particular, it first applies the feature specialization operation to all incoming edges. It then redirects all feature-specialized edges that require that the feature have a given value to one node; and all the feature-specialized edges that require that the feature not have that value to the other node. This modification operation allows the transformation to record history information about what combinations of output tags and input features have been encountered so far.

```
1. Initialize candidates to be the singleton set containing the identity transducer.
2. Repeat ...
     • When a cluster node is available:
         (a) Generate a new FST, by applying a randomly selected modification operation to a random member of the candidates set.
         (b) Use the cluster node to evaluate the new FST (as described in Section 4.3).
     • When a cluster node has finished evaluating an FST:
         (a) Add the new FSTs to the candidates set.
         (b) Sort the candidates set by their score on the held-out data, and discard all but the fifteen highest-scoring candidates.
   ... until no improvement is made for 50 consecutive iterations.
3. Return the candidate FST with the highest score.
```

Figure 4.10: **A Hill Climbing Algorithm for Optimizing Chunk Encodings**.

4.6 A Hill Climbing Algorithm for Optimizing Sequence Encodings

Having defined a set of modification operations for output-encoding FSTs, we can now use those operations to search for improved output encodings. In particular, we can use a hill-climbing approach to search the space of possible encodings for an encoding which yields increased performance. This approach starts with a simple initial FST, and makes incremental local changes to that FST until a locally optimal FST is found. In order to increase the search speed, all experiments were performed on a computer cluster, using up to 16 nodes at a time. In order to help avoid sub-optimal local maxima, we use a fixed-size beam search. A summary of this algorithm is shown in Figure 4.10.

4.7 Improving Search Speed Using Error Correcting Codes

One of the major limiting factors for this hill climbing algorithm is the speed with which we can evaluate individual FSTs. Over the course of the algorithm, we need to evaluate hundreds of different FSTs; and for each FST, we need to train and run a separate CRF model. Unfortunately, CRFs can take a long time to train. What's more, the the time that it takes to train a CRF is proportional to the square of the number of labels used by the CRF; but many of the transformations we are interested in can significantly increase the size of the label set by repeatedly subdividing labels.

We therefore employed a method described by Cohn, Smith, and Osborne, which decreases the time required to train a multi-label CRF by approximating it using a collection of binary CRFs (Cohn et al., 2005). This method begins by defining a mapping from each label L_i to a unique n-bit string "code string" C_i. It then generates n relabeled variants of the training corpus, one for each bit in the code string. Each relabeled variant j is formed by replacing each output label L_i with the label "1" (if $C_{ij} = 1$) or "0" (otherwise). These n relabeled training corpora are then used to train n independent CRF models. Cohn et al. explore a variety of decoding techniques, but the most successful defines the score for a labeled sequence to be the product of the scores that the n independent CRFs assign to that sequence. This product can be calculated efficiently using a variant of the Viterbi algorithm.

Unfortunately, this transformation from a multi-label CRF to a set of binary CRFs makes it impossible to make use of subset features, which we have seen can be important for allowing the model to make generalizations over groups of related labels. However, by choosing code strings based on the same information that was used to form the subset features, we can ensure that we include both specialized models (which can capture idiosyncrasies of different subdivided labels) and more general models (which can generalize over these differences, where possible). In

Label	Code String								
	C_{i1}	C_{i2}	C_{i3}	C_{i4}	C_{i5}	C_{i6}	C_{i7}	C_{i8}	C_{i9}
I_1	1	0	0	1	0	0	0	0	0
$I_{2,1}$	1	1	0	0	1	0	0	0	0
$I_{2,2}$	1	1	0	0	0	1	0	0	0
O_1	0	0	1	0	0	0	1	0	0
O_2	0	0	1	0	0	0	0	1	0
B	0	0	0	0	0	0	0	0	1

Figure 4.11: **Error Correcting Output Code Example**. This table shows the set of error correcting output codes that would be used for an output encoding that uses the following tag set: $\{I_1,\ I_{2,1},\ I_{2,2},\ O_1,\ O_2,\ B\}$. An independent learner is trained for each bit of the code string.

particular, we use the structure of the subdivided labels to define the set of code strings. Each code string bit corresponds to a single subdivided label *or* a single prefix to a subdivided label. In other words, for each subset feature group that we would have used in the original (multi-label) CRF, we add a single corresponding bit to the code string. For example, Figure 4.11 shows the error correcting output codes that would be used for an encoding that uses the tag set $\{I_1,\ I_{2,1},\ I_{2,2},\ O_1,\ O_2,\ B\}$. Note the similarity of this Figure to Figure 4.6, which describes the set of subset feature groups that would be used for this same tag set.

4.7.1 Improving Search Speed by Reusing Binary Models

The original motivation behind the application of error correcting output codes to CRFs was to reduce the training time for problems with large label spaces, since training time is proportional to the square of the number of labels. But this method has a substantial further benefit when used in the context of our hill-climbing algorithm, which needs to evaluate a large number of closely related output encodings. In particular, most of the FST modification operations only affect a small subset of the output labels generated by the FST. As a result, most of the individual binary models that we have already trained can simply be reused as-is when evaluating the new FST system. Thus, for each FST that we evaluate, we only need to train a few new binary CRFs (typically 2-5), rather than one for each bit in the code string.

In order to implement this optimization, we take the MD5 hash of the training corpus used to train each binary CRF. Before training a new binary CRF, we compare the MD5 hash of its training corpus to the hashes of the corpora for models we have already trained. If any hash matches, the we check to make sure that the training corpora are indeed identical (to avoid the very unlikely case where there's a hash collision); and if so, we simply re-use the model that we've already trained.

Between the improvement in training speed obtained by using the basic error correcting output code technique, and the improvement obtained from re-using individual binary models, we can increase the training speed for the hill climbing system by a factor of anywhere from 8x to 40x (depending on the size of the canonical tag set). This allows us to explore the search space of output encodings much more quickly and effectively.

4.8 Hill Climbing Experiments

In order to evaluate how effectively the hill-climbing algorithm can find improved output encodings, I tested the system on three different sequence prediction tasks: noun phrase chunking, part-of-speech tagging, and bio-entity recognition.

4.8.1 Noun Phrase Chunking

Training and testing for the noun phrase chunker were performed using the noun phrase chunking corpus described in Ramshaw & Marcus (1995) (Ramshaw and Marcus, 1995). A randomly selected 10% subset of the original training corpus was used as held-out data, to provide feedback to the hill-climbing system. The IOB1 encoding used by the corpus files was used as the canonical encoding. All models were trained using the method described in Section 4.7. All models use the same

System	F_1 (Held-out)	F_1 (Test)
Canonical encoding	94.83	93.69
Learned encoding	96.32	94.56 ⋆

Table 4.3: **Results for the NP Chunking Experiment**.
⋆: score is significantly different from the baseline score.

set of features that were used to evaluate the hand-crafted encoding transformation described in Section 4.2; these features are listed in in Figure 4.3.

After 400 iterations, the hill-climbing system increased chunking performance on the held-out data from a F-score of 94.8 to an F-score of 96.3. This increase was reflected in an improvement on the test data from an F-score of 93.7 to an F-score of 94.6.

The final learned FST is quite complex, with 14 states and a total of 52 tags. Much of the performance improvement that this FST achieved comes from subdividing tags to encode information about the recent history of tags that have been predicted. These transformations are similar to the transformations that can be used to move to a second or third order Markov model; but they are selective about which history contexts should be expanded, and which should not. The learned FST also contains feature specializations based on the part-of-speech feature, which create special tags for words that are proper nouns, commas, and conjunctions. In addition, the learned FST contains six other highly specific feature specializations, which are most likely a result of overfitting on the heldout corpus.

4.8.2 Part of Speech Tagging

Training and testing for the part of speech tagger were performed using the Penn Treebank II corpus. Training was performed using sections 2-21 of the Treebank, with a randomly selected 10% subset of the training corpus used as held-out data, to provide feedback to the hill-climbing system. Tagging accuracy was evaluated using section 24 of the Treebank. All models were trained using the method described in

Feature	Description
y_i	The current output tag.
y_i, w_{i+n}	A tuple of the current output tag and the $i{+}n$th word, $-2 \leq n \leq 2$.
$y_i, \text{len}(w_i)$	A tuple of the current output tag and the length of the current word.
$y_i, \text{prefix}_n(w_i)$	A tuple of the current output tag and the n-letter prefix of the current word, $1 \leq n \leq 4$.
$y_i, \text{suffix}_n(w_i)$	A tuple of the current output tag and the n-letter suffix of the current word, $1 \leq n \leq 4$.
$y_i, \text{start-cap}(w_i)$	A tuple of the current output tag and a boolean that is true if the current word starts with a capital letter.
$y_i, \text{start-num}(w_i)$	A tuple of the current output tag and a boolean that is true if the current word starts with a number.
$y_i, \text{has-cap}(w_i)$	A tuple of the current output tag and a boolean that is true if the current word contains a capital letter.
$y_i, \text{has-num}(w_i)$	A tuple of the current output tag and a boolean that is true if the current word contains a number.
$y_i, \text{has-dash}(w_i)$	A tuple of the current output tag and a boolean that is true if the current word contains a hyphen.

Figure 4.12: **Feature Set for the CRF Part of Speech Tagger**. y_i is the i^{th} output tag; w_i is the i^{th} word.

Section 4.7. All models use the feature set described in Figure 4.12.

After 150 iterations, the hill-climbing system increased part of speech tagging accuracy on the held-out data from 95.8 to 96.6. This increase was reflected in an improvement in accuracy on the test data from 95.9 to 96.4. As was the case for the NP chunker, most of the improvement came from moving to a “mixed-order” tagger, which uses subdivided tags to record varying amounts of information about the recent history of tags that have been predicted.

System	Accuracy (Held-out)	Accuracy (Test)
Canonical encoding	95.81	95.94
Learned encoding	96.62	96.41 ⋆

Table 4.4: **Results for the Part of Speech Tagging Experiment**.
⋆: score is significantly different from the baseline score.

System	F_1 (Held-out)	F_1 (Test)
Canonical encoding	67.82	67.56
Learned encoding	70.14	69.98 ⋆

Table 4.5: **Results for the Bio-Entity Recognizer Experiment**.
⋆: score is significantly different from the baseline score.

4.8.3 Bio-Entity Recognition

Training and testing for the bio-entity recognizer were performed using the JNLPBA bio-entity recognition corpus, which was derived from the Genia corpus and distributed for the JNLPBA shared task in bio-entity recognition (Kim et al., 2004). The IOB2-based encoding used by the corpus files (described in Section 4.1.3) was used as the canonical encoding. All models were trained using the method described in Section 4.7. All models use the feature set described in Figure 4.13.

After 250 iterations, the hill-climbing system increased bio-entity tagging performance on the held-out data from a F-score of 67.8 to an F-score of 70.1. This increase was reflected in an improvement on the test data from an F-score of 67.6 to an F-score of 70.0.

4.9 Hill Climbing with HMMs

As was discussed in Section 4.7, a major limiting factor for the hill climbing algorithm is the speed with which we can evaluate the effect of individual FST transformations on system performance. Although the methods described in that section significantly increased the speed with which we can evaluate FSTs, the fact remains that even binary CRFs are relatively slow to train.

Feature	Description
y_i	The current output tag.
y_i, w_{i+n}	A tuple of the current output tag and the $i + n$th word, $-2 \leq n \leq 2$.
y_i, to-lower(w_i)	A tuple of the current output tag and a case-normalized copy of the current word.
y_i, is-number(w_i)	A tuple of the current output tag and a boolean that is true if the current word is a roman numeral.
y_i, is-roman(w_i)	A tuple of the current output tag and a boolean that is true if the current word is a roman numeral.
y_i, is-dna(w_i)	A tuple of the current output tag and a boolean that is true if the current word is a composed entirely of the letters "G", "C", "A", and "T".
y_i, is-rna(w_i)	A tuple of the current output tag and a boolean that is true if the current word is a composed entirely of the letters "G", "C", "A", and "U".
y_i, word-shape(w_i)	A tuple of the current output tag and the "word shape" of the current word, which is formed by replacing each upper-case letter with "A"; each lower-case letter with "a"; each digit with "0"; and each non-alphanumeric character with "-".
y_i, simple-word-shape(w_i)	A tuple of the current output tag and the "simplified word shape" of the current word, which is formed from the word shape by collapsing any consecutive sequences of the same shape character.
y_i, len(w_i)	A tuple of the current output tag and the length of the current word.
y_i, prefix$_n$(w_i)	A tuple of the current output tag and the n-letter prefix of the current word, $1 \leq n \leq 4$.
y_i, suffix$_n$(w_i)	A tuple of the current output tag and the n-letter suffix of the current word, $1 \leq n \leq 4$.
y_i, has-dash(w_i)	A tuple of the current output tag and a boolean that is true if the current word contains a hyphen or dash.
y_i, has-cap(w_i)	A tuple of the current output tag and a boolean that is true if the current word contains a capital letter.
y_i, has-num(w_i)	A tuple of the current output tag and a boolean that is true if the current word contains a number.

Figure 4.13: **Feature Set for the CRF Bio-Entity Recognizer**. y_i is the i^{th} output tag; w_i is the i^{th} word.

I therefore performed a series of experiments to examine the effectiveness of the hill climbing algorithm when used with Hidden Markov Models (HMMs), rather than linear chain CRFs. Unlike CRFs, which must be trained using global optimization techniques, HMMs can be trained based on a single pass through the data. As a result, the hill climbing algorithm can evaluate the effect of an FST transform for an underlying HMM model many times faster than it can for a CRF model. It is therefore possible to explore a much larger portion of the search space of potential transformations.

However, HMMs do not have any analogue to subset features (which can be used to generalize over subdivided labels). They therefore can not take advantage of the relationships between labels that are expressed by the FST when it subdivides a label. Instead, it treats each label (including subdivided labels) as a simple unique symbol.

For my first experiment, I used the hill-climbing algorithm to search for optimal encodings for the noun phrase chunking task, using HMMs as the underlying sequence prediction model. In order to allow the HMMs to use the same feature set that was used by the CRF models, I used an HMM variant where each state generates multiple emissions – one for each feature. I used the same training and testing corpora, canonical encoding, and feature set that were used by the CRF model described in Section 4.8.1.

After 1000 iterations, the hill-climbing system increased chunking performance on the held-out data from a F-score of 87.8 to an F-score of 91.8. This increase was reflected in an improvement on the test data from an F-score of 87.9 to an F-score of 91.5. (See Table 4.6.) However, almost all of this improvement in performance is achieved by the 20th iteration, which produced an FST that encodes chunks using a variant of of the second-order IOB2 encoding. This FST achieves a score of 91.4 on the both the held-out corpus and the test corpus.

Although the hill-climbing system is able to improve the performance of the

System	F_1 (Held-out)	F_1 (Test)	
Canonical encoding	87.78	87.86	
Best encoding after 20 iterations	91.44	91.41	⋆
Best encoding after 100 iterations	91.69	91.52	⋆
Best encoding after 1000 iterations	91.82	91.56	⋆

Table 4.6: **Results for HMM Hill Climbing Experiment (1)**. The performance of the best output encoding found after 1000 iterations, using the full feature set and no back-off.

⋆: score is significantly different from the baseline score.

original HMM, most of this benefit comes from transformations that move it to a second-order model. The hill-climbing system is able to make very few incremental changes beyond that point. What's more, even after 1,000 iterations, the hill-climbing system is unable to generate a sequence labeler that comes anywhere close to the performance of a CRF.

One potential explanation for the lack of improvement comes from the fact that the HMM model makes a strong independence assumption that all emissions are conditionally independent, given the tag sequence. However, this is clearly not true, given that many of the features used by the HMM (described in Figure 4.3) exhibit a high degree of mutual dependence. I therefore ran a second experiment, using just two features: the current word (w_i) and the current word's part of speech tag (t_i). The results of this second experiment are shown in Table 4.7. By moving to a simpler feature set, we significantly reduced the performance of the HMM. However, the hill-climbing algorithm was able to overcome some of that performance drop by building fairly complex encoding transformations.

Another potential explanation for the lack of improvement in the original experiment (beyond that provided by moving to a second-order model) is that HMMs are unable to capture the generalizations that are expressed by subdivided labels. In order to allow the HMM model to capture these generalizations, I tried smoothing the emission probability distributions by adding backoff distributions based on the way in which labels are subdivided. For example, I constructed a smoothed probability

System	F_1 (Held-out)	F_1 (Test)
Canonical encoding	82.21	82.16
Best encoding after 100 iterations	85.72	85.08 $\star$
Best encoding after 1000 iterations	87.24	87.12 $\star$

Table 4.7: **Results for HMM Hill Climbing Experiment (2)**. The performance of the best output encoding found after 1000 iterations, using a reduced feature set and no back-off.

$\star$: score is significantly different from the baseline score.

System	F_1 (Held-out)	F_1 (Test)
Canonical encoding	87.78	87.86
Best encoding after 100 iterations	89.12	88.94 $\star$
Best encoding after 1000 iterations	90.06	89.59 $\star$

Table 4.8: **Results for HMM Hill Climbing Experiment (3)**. The performance of the best output encoding found after 1000 iterations, using the full feature set and back-off.

$\star$: score is significantly different from the baseline score.

distribution $\hat{P}(w_i = the|y_i = B_{2,1})$ by averaging the three probability distributions $P(w_i = the|y_i = B_{2,1})$, $P(w_i = the|y_i = B_{2,*})$, and $P(w_i = the|y_i = B_*)$. For my third experiment, I tried running the hill-climbing algorithm using the full feature set (as used in the first experiment), but using backoff to smooth the distributions learned by the HMM. The results of this experiment are shown in Table 4.8. Unfortunately, the addition of backoff distributions did not improve performance any.

4.9.1 Hill Climbing with HMMs: Discussion

We have seen that the hill-climbing algorithm can improve the performance of an HMM by transforming its output structure, especially during the first hundred iterations. Most of these improvements come from subdividing the original label set in ways that allow the HMM's tags to capture more information about the recent history of tags that have been generated. These transformations are similar to the transformations that can be used to express higher-order HMMs; except that rather than extending the history window for all tags equally, these transforms can selectively extend the history window for different tag sequences.

However, beyond these basic history-encoding transformations, the hill-climbing algorithm is able to improve the performance of the HMM very little. One likely explanation of this limitation is the fact modification operations tend to create novel output labels, reducing the amount of training data that is available to train the probability distributions associated with each tag.

Another potential explanation of this limitation comes from a difference in the way that HMMs and CRFs are trained. CRFs are trained by global optimization techniques, allowing them to choose the set of model parameters that yield the best overall performance. This can allow the CRF model to adapt to any negative effects caused by changes to the model structure. In contrast, HMMs are trained by calculating transition and emission probability distributions, independently of one another. If a transformation causes these probability distributions to become distorted, then the HMM has no way to overcome those distortions.

This analysis suggests two conditions on the underlying learning method which may be beneficial in general when applying automatic techniques to find good output encodings:

1. The underlying method should have a mechanism for handling subdivided labels that allows it to capture the important distinctions between the subdivided labels while also generalizing over their shared behavior.
2. The underlying method should be trained by global optimization.

Chapter 5

Capturing Long-Distance Constraints in a Joint Model for Semantic Role Labeling

Transforming the output encoding can improve performance in two ways. First, as was highlighted in Chapter 3, the transformation can introduce more coherent labels for local sub-problems, allowing these sub-problems to be modelled more accurately. Second, as we saw in Chapter 4, the transformation can modify the type of information that is shared between different sub-problems, affecting the type of dependencies that can be learned between those sub-problems.

This second type of effect can be further subdivided into two cases. First, the transformation can augment tags with information about the values that have been predicted for other sub-problems within a fixed history window. An example of this case is the transformation that transforms a first-order Markov model into a second-order model by replacing individual tags with pairs of adjacent tags. Second, the transformation can allow tags to propagate information about values that have been predicted arbitrarily far away. This can allow the model to capture certain types of long-distance dependencies that it was unable to capture before the transformation.

For the sequence prediction tasks discussed in Chapter 4, almost all of the improvements in performance resulted from transformations that made local subproblems more coherent; and transformations that added history information over a fixed window size. One possible explanation for this fact is that the three sequence prediction problems we considered were relatively "local" problems – in particular, the number of cases where long-distance constraints were important in deciding the correct output value were small enough that they had very little effect on the overall performance scores.

In this Chapter, we therefore look at a task in which long-distance dependencies are more important: Semantic Role Labeling (SRL). The system that we will consider in this Chapter differs from the SRL system we explored in Chapter 3 in that it performs SRL using a single joint model to find and classify all of a given verb's arguments. In contrast, the SRL system discussed in Chapter 3 considers each candidate argument phrase independently of all other phrases, and attempts to find the most likely status of that phrase, without taking into account any possible interactions or constraints between different phrases in the sentence.

5.1 Local SRL Models

Much of the previous work on statistical models for semantic role labeling has divided this task down into two sub-tasks: finding likely arguments for a verb (the Id task), and determining their semantic role (the Cls task) (Gildea and Jurafsky, 2002; Yi and Palmer, 2005). Furthermore, the problem of finding and identifying each argument is usually treated independently for each potential argument.

This division of the overall SRL task into a set of local classification tasks prevents SRL models from learning important joint information, such as the fact that some argument combinations are much more likely than others, and that arguments typically do not overlap one another. Linguistic theory indicates that "thematic frames," or specific configurations of arguments, are important for defining valid argument assignments.

A number of existing systems account for these long-distance constraints and dependencies between arguments by first running a classifier that selects arguments independently; and then using a re-ranking system on the n best outputs (Gildea and Jurafsky, 2002; Pradhan et al., 2004; Thompson et al., 2003; Sutton and McCallum, 2005; Toutanova et al., 2005). Although such systems are able to take advantage of joint features, they are potentially limited by the fact that the correct labeling may not be with in the n best outputs, for any given n.

5.2 Baseline Model

Our baseline system performs three steps, in sequence. First, it applies an "argument filter" to decide which parse tree constituents should be considered for a given predicate (Figure 5.1). Second, it uses a Maximum Entropy classifier to perform the Id task, tagging each candidate constituent as either Arg or NotArg. Finally, it uses a second Maximum Entropy classifier to perform the Cls task, labeling any arguments that were labeled Arg with their argument identifier. The tag set for this second classifier is shown in Figure 5.2. The features used for both classifiers are shown in Figure 5.3. This set of features, which was selected based on a survey of the features that have been reported to work well in other SRL systems, will be used for all the SRL models discussed in this Chapter.

When evaluated on PropBank corpus (section 23), using Dan Bikel's parser (Bikel, 2004a), this baseline model achieves an F_1 score of 74.9 (precision=78.1; recall=72.0).

Return any candidate constituents c for a predicate p if either:

1. c's parent is an ancestor of p; and
   c is not punctuation or a preposition; and
   c is not an ancestor of p.

2. c's grandparent is an ancestor of p; and
   c's parent is a PP, S, NP, or ADVP; and
   c is not punctuation or a preposition; and
   c is not an ancestor of p.

Figure 5.1: **SRL Argument Filter**. The joint SRL system creates a linear sequence of constituents to tag by first taking a depth first traversal of all constituents; and then pruning out any constituents that do not meet the criteria listed here.

Numbered Arguments
Arg0
Arg1
Arg2
Arg3
Arg4
Arg5

Modifier Arguments	
ArgM-Adv	ArgM-Mod
ArgM-Cau	ArgM-Neg
ArgM-Dir	ArgM-Pnc
ArgM-Dis	ArgM-Prd
ArgM-Ext	ArgM-Rec
ArgM-Loc	ArgM-Tmp
ArgM-Mnr	

Figure 5.2: **SRL Tag Set for the Cls Task**.

Id feature predicates	Cls feature predicates	
Head	Head	LastWord
HeadPos	HeadPos	LeftCat+Parent
Path	Path	LeftCat+Parent+Cat
PrjPath	PrjPath	Voice
TDist	TDist	SubCat
Verb+Head	Verb+Head	Verb+Cat
Verb+Path	Verb+Path	Cat
Verb+Prep+Head	Verb+ Prep+Head	Pos
Prep+Head	Prep+Head	Voice+Pos+Cat
Verb+Prep	Verb+Prep	
Verb+Cat	Verb	
WDist	FirstWord	

Figure 5.3: **SRL Features**. Feature predicates and predicate combinations used by Id and Cls models. The basic predicates are defined in Figure 5.4.

Predicate	Definition
Verb	stemmed predicate
Head	stemmed head of the phrase
HeadPos	part of speech of head of the phrase
Cat	syntactic category of the phrase
Path	path from phrase to predicate
Pos	phrase before or after predicate?
Voice	voice of predicate (active/passive)
SubCat	CFG expansion of predicate's parent
First	First word of phrase
Last	Last word of phrase
Prep	Preposition (for PP args)
WDist	Word distance
TDist	Tree distance
LeftCat	category of phrase's left sibling
Parent	category of phrase's parent
PrjPath	path from phrase's maximal projection to predicate

Figure 5.4: **Definition of SRL Feature Predicates.**

Non-Argument
NotArg

Numbered Arguments
Arg0
Arg1
Arg2
Arg3
Arg4
Arg5

Modifier arguments	
ArgM-Adv	ArgM-Mod
ArgM-Cau	ArgM-Neg
ArgM-Dir	ArgM-Pnc
ArgM-Dis	ArgM-Prd
ArgM-Ext	ArgM-Rec
ArgM-Loc	ArgM-Tmp
ArgM-Mnr	

Figure 5.5: **SRL Tag Set for the Joint Id and Cls Model**.

5.3 A Joint Id and Cls model

As a first step towards building an SRL model that assigns all argument labels jointly, I created a model that performs the Id and Cls tasks jointly. Given a parse tree constituent, this model tags it with one of the 19 labels shown in Figure 5.5.

The simplest way to create this joint Id and Cls model would be to take the union of the feature predicates used by the two individual models, and combine each feature predicate g_i with each possible label l_j:

System	Precision	Recall	F_1
Separate Id & Cls	78.1	72.0	74.9
Joint Id & Cls (simple features)	77.6	71.4	74.4

Table 5.1: **Performance of the Joint Id & Cls Model w/ Simple Features**. Performance evaluated on the PropBank corpus (Section 23), using automatic parses generated by Bikel's parser.

$$f_k(x, y) = \begin{cases} 1 & \text{if } g_i(x) = 1 \text{ and } y = l_j \\ 0 & \text{otherwise} \end{cases} \tag{5.1}$$

$$g_i \in G_{\text{ID}} \cup G_{\text{CLS}} \tag{5.2}$$

$$l_i \in \{\textsc{NotArg}, \textsc{Arg0}, ..., \textsc{ArgM-Tmp}\} \tag{5.3}$$

However, as shown in Table 5.1, this simple joint Id and Cls model has worse performance than a system that performs Id and Cls separately. There are two primary reasons for this. First, the set of features that are most useful for the Id task are not identical to the set of features that are useful for the Cls task (Xue and Palmer, 2004). And second, by treating the Id task separately, the Id model is able to combine training data from different argument types, which helps to reduce the sparse data problem. This is important, because the Id task contributes significantly to the overall system performance (Toutanova et al., 2008).

However, we can take advantage of the fact that Maximum Entropy features may be defined as arbitrary functions on x and y to overcome the two issues that degrade the performance of simple joint Id and Cls models. In particular, we define two groups of features, one targeted at helping make the Id distinction, and the other targeted at helping make the Cls distinction. This approach is analogous to the definition of "subset features" in Chapter 3.

These two groups of features differ from one another in two ways. First, they each use different sets of feature predicates $g(x)$. This allows us to separate out the features that are useful for the Id task from features that are useful for the Cls task.

Second, the two feature groups differ in how their feature functions depend on the output label y.

We form features from each ID predicate $g^{\text{ID}}(x)$ as follows:

$$f(x, y) = \begin{cases} 1 & \text{if } y = \textsc{NotArg} \text{ and } g^{\text{ID}}(x) = 1 \\ 0 & \text{otherwise} \end{cases} \tag{5.4}$$

$$f(x, y) = \begin{cases} 1 & \text{if } y \neq \textsc{NotArg} \text{ and } g^{\text{ID}}(x) = 1 \\ 0 & \text{otherwise} \end{cases} \tag{5.5}$$

In other words, we define one feature that fires if y is not an argument, and a second feature if y is *any* argument. By defining features that group all the non-NotArg labels together, we avoid the sparse data issue that can decrease the performance for a simple joint model.

Similarly, we form features from each Cls predicate g^{Cls} by pairing it with each label l *except* NotArg:

$$f(x, y) = \begin{cases} 1 & \text{if } y = l \text{ and } g^{\text{Cls}}(x) = 1 \\ 0 & \text{otherwise} \end{cases} \tag{5.6}$$

This allows the model to use these features to distinguish the different argument labels from one another, but forces it to use the ID features to decide whether a phrase is an argument or not.

Table 5.2 compares the performance of this joint ID & Cls model with the baseline model. As we can see, the use of complex features allows the model to overcome the issues that degraded performance when we built a joint model using simple features. The improvement in performance relative to the baseline model indicates that the joint model is able to take advantage of dependencies between the ID and the Cls sub-tasks.

System	**Precision**	**Recall**	F_1
Separate Id & Cls	78.1	72.0	74.9
Joint Id & Cls (simple features)	77.6	71.4	74.4
Joint Id & Cls (complex features)	78.3	72.6	75.3

Table 5.2: **Performance of the Joint Id & Cls Model w/ Complex Features**. Performance evaluated on the PropBank corpus (Section 23), using automatic parses generated by Bikel's parser.

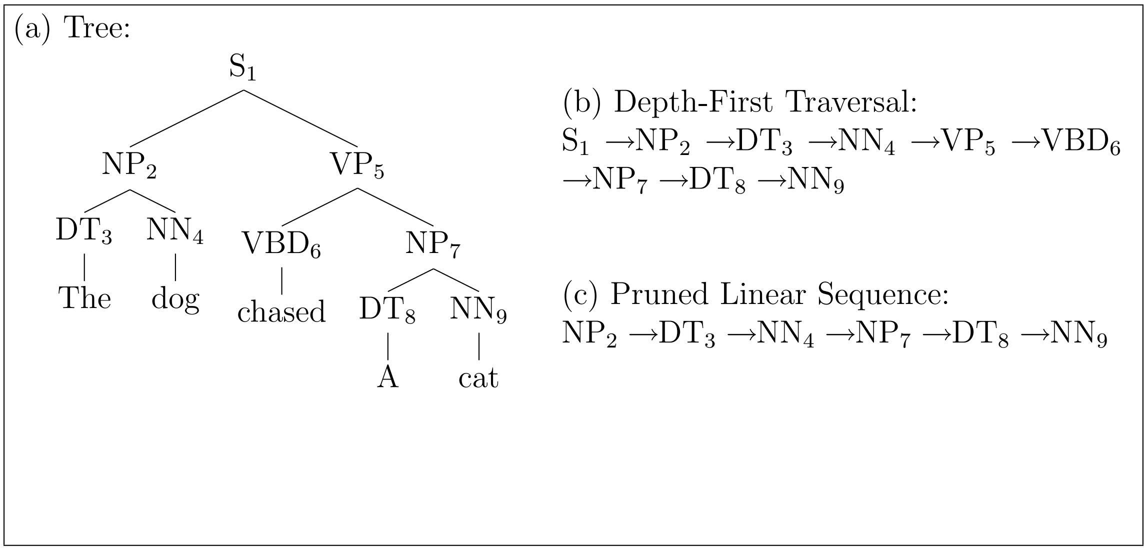
5.4 A Joint Sequence Model for SRL

This joint Id & Cls model lets us take advantage of dependencies between the Id and Cls sub-tasks; but it still considers the classification of each phrase as an independent task. In this section, we develop a model that allows us to capture long-distance dependencies between classification decisions for different arguments. These dependencies between arguments carry important information, which should help us to improve the overall SRL output.

In order to make use of the framework we have developed in Chapter 4, we will frame the semantic role labeling task as a sequence classification task. The most straight-forward way to design a sequence-classifying SRL system would be to tag each word with a label, indicating whether it is part of an argument; and if so, what its tag is. However, prior work has shown that the constituent structure provided by a parse tree is very helpful in performing accurate SRL (Gildea and Jurafsky, 2002; Yi and Palmer, 2005; Toutanova et al., 2005).

I therefore chose to design an SRL system that tags constituents, rather than words. The constituents are arranged into a linear sequence by taking a pruned depth-first traversal of the tree. Figure 5.6 shows the traversal order for an example sentence. Constituents are pruned using the same "argument filter" algorithm that is used by the baseline model to select candidate argument phrases (Figure 5.1).

Once the parse tree has been converted to a sequence, we can use a Linear Chain Conditional Random Field (LC-CRF) to find the most likely sequence of tags for a given input.



(b) Depth-First Traversal:
$S_1 \rightarrow NP_2 \rightarrow DT_3 \rightarrow NN_4 \rightarrow VP_5 \rightarrow VBD_6 \rightarrow NP_7 \rightarrow DT_8 \rightarrow NN_9$

(c) Pruned Linear Sequence:
$NP_2 \rightarrow DT_3 \rightarrow NN_4 \rightarrow NP_7 \rightarrow DT_8 \rightarrow NN_9$

Figure 5.6: **SRL as a Sequential Task**. The parse tree (a) is converted to a linear sequence (b) by taking a depth-first traversal of its constituents. This linear sequence is then pruned by removing constituents that contain the predicate (c).

For our canonical encoding, we use the set of tags that was used by the joint Id & Cls model (Figure 5.5). For example, the correct sequence of canonical tags for the example shown in Figure 5.6 is:

NP_2	DT_3	NN_4	NP_7	DT_8	NN_9
Arg0	NotArg	NotArg	Arg1	NotArg	NotArg

5.4.1 CRF Sequence Model w/ Canonical Encoding

We begin by considering the performance of a CRF model that uses the canonical encoding directly. Table 5.4.1 compares the performance of this CRF to the performance of the Joint Id and Cls model. The performance is essentially unchanged. This should not come as a surprise, because the Markov assumptions made by the CRF only allow the model to learn dependencies between adjacent tags. In a depth-first traversal of the tree, the previous tag for a node is the tag assigned to the rightmost descendant of its left sister (and *not* the tag of its left sister). As a result, the previous tag will usually be NotArg. In essence, this CRF model is too local

Model	Precision	Recall	F_1
Joint Id & Cls (complex features)	78.3	72.6	75.3
CRF (canonical encoding)	78.2	72.7	75.3

Table 5.3: **Performance of the CRF Sequence Model w/ Canonical Encoding**. Performance evaluated on the PropBank corpus (Section 23), using automatic parses generated by Bikel's parser.

to capture any useful dependencies between argument labels.

One possible solution would be to move to a higher-order Markov model, which would allow the CRF to learn slightly longer dependencies. However, this will tend to fragment the training data, causing issues with sparse data and overfitting; and will not yield much benefit, because many of the important dependencies span over more than just two or three nodes in the depth-first traversal.

5.5 Applying the Hill Climbing System to the Joint Sequence SRL Model

Instead, we can use the hill climbing algorithm developed in Chapter 4 to search for encoding transformations that will allow the CRF to make use of an appropriate history context. Starting with an identity FST that maps the canonical encoding to itself, this algorithm uses a variety of FST-modification operations to generate novel encodings, and evaluates those encodings using a heldout data set (section 24 of the PropBank corpus).

I ran the hill-climbing system five times, each time for 150 iterations. The results of these five runs are shown in Table 5.4. Each run improved the performance of the SRL system, with absolute improvement on the test corpus ranging from +0.5% to +0.9% in F_1 score.

Each run of the SRL system arrived at a somewhat different encoding; however there were many similarities between the five encodings generated by the five runs. In

System	F_1 (Held-out)	F_1 (Test)
Canonical encoding	74.8	75.3
Learned encoding (Run 1)	76.3	76.1 $\star$
Learned encoding (Run 2)	76.2	75.9 $\star$
Learned encoding (Run 3)	76.3	76.2 $\star$
Learned encoding (Run 4)	76.4	75.8 $\star$
Learned encoding (Run 5)	76.2	76.1 $\star$

Table 5.4: **Hill-Climbing Results for the CRF SRL Model**.
$\star$: score is significantly different from the baseline score.

order to gain some insight into the generated encodings, I examined the transformations they made by hand, paying special attention to transformations that resulted in larger improvements to the held-out score. I concluded that the most dramatic improvements came from transformations that added history information about what non-NotArg tags had been predicted, combined with information about whether the constituent being considered occurs before or after the predicate. Larger gains were obtained from transforms that captured more common history patterns.

This result is intuitively plausible. In particular, by capturing information about the history of arguments that have been predicted, the model can make use of information about which arguments tend to occur together, and in which combinations. In other words, the model can learn patterns over argument frames.

However, the automatically discovered encodings were only able to make use of history information within a fairly limited window. This can be explained by the fact the the encoding transformations that are most easily reachable from the initial (identity) FST all make use of history information from limited windows. Although it is possible to arrive at encoding transformations that maintain history information over unlimited distances, it takes significantly more FST modification steps to arrive at them.

5.6 Generalizing the Hill Climbing Results

Based on this analysis of the automatically generated encoding transformations, I created a hand-crafted encoding transformation that generalizes their history-recording behavior.

This hand-crafted transformation augments each tag with information about the ordered set of argument tags that have been predicted, including the location of the predicate. Since Arg0-Arg5 (core) arguments are much more important in defining thematic frames than ArgM (modifier) arguments, we only include history information about Arg0-Arg5 arguments.

5.6.1 The Augmented Tag Set

In particular, the hand-crafted transformation replaces each tag y_i with a tuple $\langle h_i, y_i \rangle$, where h_i is a history value summarizing the set of tags $y_0 ... y_{i-1}$ for the tokens that precede the i^{th} token. Since we are mostly concerned with the relative position of the numbered arguments and the predicates, h_i is defined to keep track of only the tags corresponding to numbered arguments and the predicate. In particular, we define h_i as follows, where $pred(i)$ is true iff x_i is the first constituent that follows the predicate:

$$h_1 = \begin{cases} \langle \textsc{Start} \rangle & \text{if } pred(i) \\ \langle \textsc{Start}, \textsc{Pred} \rangle & \text{if } \neg pred(i) \end{cases} \tag{5.7}$$

$$\forall i > 1 : h_i = \begin{cases} h_{i-1} + \langle y_{i-1}, \textsc{Pred} \rangle & \text{if } y_{i-1} \in \textsc{Arg}0\text{-}5 \wedge pred(i) \\ h_{i-1} + \langle y_{i-1} \rangle & \text{if } y_{i-1} \in \textsc{Arg}0\text{-}5 \wedge \neg pred(i) \\ h_{i-1} + \langle \textsc{Pred} \rangle & \text{if } y_{i-1} \notin \textsc{Arg}0\text{-}5 \wedge pred(i) \\ h_{i-1} & \text{if } y_{i-1} \notin \textsc{Arg}0\text{-}5 \wedge \neg pred(i) \end{cases} \tag{5.8}$$

For example, the tag $\langle \textsc{Arg}0, \textsc{Pred}, \textsc{Arg}1 \rangle$ would be used for a constituent filling the Arg1 role, and following both an Arg0 argument and the predicate.

History	Frequency
⟨Start⟩	25.4
⟨Start, Arg0⟩	16.1
⟨Start, Arg0, Pred⟩	16.1
⟨Start, Arg0, Pred, Arg1⟩	14.6
⟨Start, Pred⟩	04.7
⟨Start, Arg1⟩	04.1
⟨Start, Pred, Arg1⟩	04.0
⟨Start, Arg1, Pred⟩	04.0
⟨Start, Arg1, Pred, Arg2⟩	01.6
⟨Start, Arg0, Pred, Arg1, Arg2⟩	01.5
⟨Start, Pred, Arg1, Arg0⟩	01.1
⟨Start, Pred, Arg1, Arg2⟩	0.8
⟨Start, Arg0, Pred, Arg2⟩	0.8
⟨Start, Arg2⟩	0.5
⟨Start, Arg2, Pred⟩	0.5
⟨Start, Arg2, Pred, Arg1⟩	0.5
⟨Start, Arg0, Pred, Arg2, Arg1⟩	0.4
⟨Start, Arg1, Pred, Arg2, Arg4⟩	0.3
⟨Start, Pred, Arg2⟩	0.2
⟨Start, Pred, Arg0⟩	0.2

Figure 5.7: **20 Most Frequent History Values**. These 20 prefixes are combined with the canonical tag set to form the augmented tag set used by the hand-crafted encoding transformation.

At first glance, this tag set may seem unreasonable. In particular, there are at least 2,000 possible values for h_i; and pairing these with the 19 canonical tags gives us a tag set size of almost 40,000.[1] However, of the 2,000 possible history values, very few actually get used in practice; and the most frequent history values account for a vast majority of the history values that are used. I therefore pruned the full-history tag set to a more manageable tag set by only using the 20 most frequent history values. These 20 history values, shown in Figure 5.7, provide the correct history for 97% of the tags that are actually used in PropBank. Combining these 20 histories with the 19 canonical tags gives a total of 380 augmented tags.

[1] This is making the generous assumption that each argument only gets used once; without this restriction, the size of the tag set would be unbounded.

For the 3% of the constituents where the appropriate history is not in our chosen set, we choose the *closest matching history* by first collapsing any consecutive sequences of Arg0-5 tags with the same tag to a single Arg0-5 tag; and then removing Arg0-5 tags, in descending order of N, until we are left with a history in our chosen set. For example, if the previous node's tag was $\langle \textsc{Start}, \textsc{Pred}, \textsc{Arg3} \rangle$, and we wish to predict the semantic role label Arg1 for the next node, then we would first replace the previous node's tag with the closet matching history, $\langle \textsc{Start}, \textsc{Pred} \rangle$; and then combine it with the label Arg1 to form the new tag $\langle \textsc{Start}, \textsc{Pred}, \textsc{Arg1} \rangle$.

5.6.2 Subset Feature Groups

If we simply used these 380 tags directly, the CRF model would suffer from significant sparse data problems. However, we can once again make use of the fact that features are defined as general functions on both the input value and the output tag, to avoid these problems. In particular, we will use three subset feature groups, to allow the model to generalize over differing amounts of history information. The first subset feature group combines an input predicate with the complete augmented tag, including the history:

$$f(x, y) = \begin{cases} 1 & \text{if } y = l \text{ and } g(x) = 1 \\ 0 & \text{otherwise} \end{cases} \tag{5.9}$$

The second group combines an input predicate with all augmented tags that end in a given tag – i.e., it ignores the history:

$$f(x, \textit{history-tag}) = \begin{cases} 1 & \text{if } \textit{tag} = t \text{ and } g(x) = 1 \\ 0 & \text{otherwise} \end{cases} \tag{5.10}$$

The final feature group combines an input predicate with all augmented tags that end in a given pair of tags – i.e., it only considers what the most recent predicted

Model	Precision	Recall	F_1
Separate Id & Cls (baseline)	78.1	72.0	74.9
Joint Id & Cls (complex features)	78.3	72.6	75.3
CRF (canonical encoding)	78.2	72.7	75.3
CRF (avg hill-climbing result)	78.6	73.6	76.0
CRF (best hill-climbing result)	78.8	73.7	76.2
CRF (hand-crafted transform)	79.1	73.8	76.4

Table 5.5: Performance of the sequence models on the PropBank corpus (Section 23), using Bikel's parser.

argument was:

$$f(x, ...\text{-}tag_1\text{-}tag_2) = \begin{cases} 1 & \text{if } tag_1 = t_1 \text{ and} \\ & \quad tag_2 = t_2 \text{ and} \\ & \quad g(x) = 1 \\ 0 & \text{otherwise} \end{cases} \tag{5.11}$$

I found that it was important to include a Gaussian prior to prevent the CRF model from overfitting. In particular, this prior encourages the model to use features that depend on less history when possible, and to only resort to features that depend on a large amount of history when they can not be captured without that history, and when there is sufficient training data to support them.

5.6.3 Joint Sequence Model: Results

Table 5.5 compares the performance of a CRF model trained using the hand-crafted encoding transformation with the performance of the other models we have discussed in this Chapter. We can see that the augmented tag set allows the sequential model to improve performance relative to the canonical tag, set by capturing dependencies between different arguments. The augmented tag set also gives performance which is slightly higher than the best encoding that was generated by the hill-climbing system.

5.7 Discussion

We have demonstrated that using a system which predicts all of a predicate's arguments using a single joint model can out-perform a model that labels arguments independently of one another; but only if the label set used by the joint model is sufficiently rich to allow interactions between the arguments, which are typically not adjacent. This supports the claim, made in the introduction to this Chapter, that transformations that allow tags to propagate information about values that have been predicted arbitrarily far away can improve performance, by allowing the model to capture certain types of long-distance dependencies that it was unable to capture before the transformation.

We also saw that, although the hill climbing algorithm can often improve performance, it is sometimes possible to analyze the transformations that the hill climbing system makes, and to further improve performance by generalizing or simplifying those transformations.

Unfortunately, the FSTs that are generated by the hill climbing system are often very complex, and analyzing these FSTs can be a daunting task. It took a significant amount of effort to analyze the FSTs that were generated by the hill-climbing system for the SRL task, and to determine how the transformed tags are used. However, analyzing the transformed encodings can clearly be a productive endeavor. Therefore, a useful area for future work will be looking at ways to make the generated FSTs easier to analyze. For example, it might be possible to automatically track the effects that different modification operations have on the way that specific tags are used.

Another promising avenue for future work is to look at the application of encoding transformations to tree-CRF SRL systems, such as the one described in Cohn and Blunsom (2005) (Cohn and Blunsom, 2005). This model uses both tree nodes and parent/child node pairs as cliques when defining features, and performs inference using belief propagation. An advantage of this model over the linear models discussed

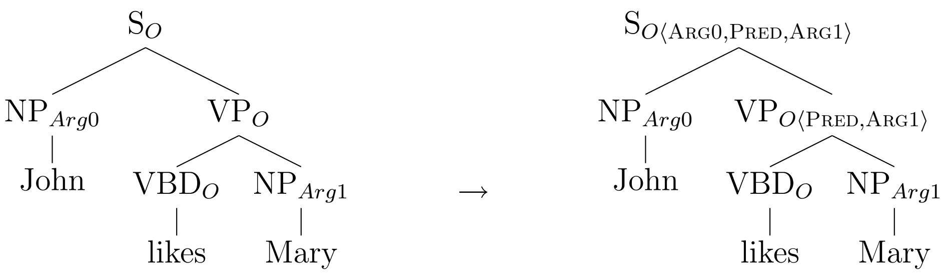


Figure 5.8: **Proposed Transform for Tree-CRF Labels**. This transform augments the “O” tag that is used for non-argument tree nodes with information about what arguments the node contains.

in this Chapter is that it does not need to ”flatten” the sequence of constituents. However, since their model uses simple tags (similar to the canonical tag set), it is unable to learn any long-distance dependencies between arguments. However, it should be possible to augment the tree-CRF's tag set to record history information, which would allow it to learn these long-distance dependencies. In particular, the “O” tag that is used for non-argument tree nodes could be augmented with information about what arguments (and in what order) are contained within that node; Figure 5.8 shows an example of this transform.

Chapter 6

Combining Multiple Learners: Global Voting

As we have seen, any given structured output prediction problem can be decomposed in many different ways. Usually, there is no single decomposition that allows us to build a model that perfectly describes the problem domain – each decomposition will give incorrect results for some set of training samples. But often, the set of misclassified training samples is different for different decompositions. We can take advantage of this fact by combining models built using different decompositions into a single model.

In simple classification tasks, complementary models are often combined using weighted voting. Using this scheme, the score assigned to each class for a given input is a combination of the individual classifier scores for that class. In this chapter, we will consider two types of voting, both of which have been used for a wide variety of classification problems: linear voting and log-linear voting.

In *linear voting*, the score assigned to each class for a given input is the weighted sum of the individual classifier scores for that class. In particular, given a set of classifier models M_i, where $M_i(y|x)$ is the score assigned by M_i to class y for input value x, and a set of weights w_i for each model M_i (s.t. $\sum_i w_i = 1$), we can define a new combined classifier $\widehat{M_+}$ as follows:

$$\widehat{M_+}(y|x) = \sum_i w_i M_i(y|x) \tag{6.1}$$

Log-linear voting is similar, except that the score assigned to each class is a weighted product, rather than a weighted sum, of the individual classifier scores for that class. In particular, given a set of classifier models M_i and weights w_i, we can define a new combined classifier $\widehat{M_\times}$ as follows:

$$\widehat{M_\times}(y|x) = \frac{1}{Z(x)} \prod_i M_i(y|x)^{w_i} \tag{6.2}$$

$$Z(x) = \sum_y \prod_i M_i(y|x)^{w_i} \tag{6.3}$$

(where $Z(x)$ is a normalization constant chosen to ensure that $\sum_y \widehat{M_\times}(y|x) = 1$.)

To use either of these combined classifiers for prediction, we simply find the class value with the highest voted score for a given input:

$$y^* = \arg\max_y \widehat{M_+}(y|x) \qquad \textit{for linear voting} \tag{6.4}$$

$$y^* = \arg\max_y \widehat{M_\times}(y|x) \qquad \textit{for log-linear voting} \tag{6.5}$$

In this chapter, we explore how linear and log-linear voting can be applied to structured output tasks. This will allow us to combine the models that we learn using various problem decompositions. Special attention is paid to sequence-prediction tasks, but many of the results that we show for sequence-prediction tasks could be generalized to other types of structured output tasks.

We begin in Section 6.1 by describing local voting techniques, and discussing how they differ from the global voting techniques that we will focus on for the remainder of the chapter. Section 6.2 then discusses the application of global linear and log-linear voting techniques to sequence prediction problems, and Sections 6.3-6.6 describe algorithms that can be used to perform voting for sequence prediction problems. Section 6.7 describes a set of experiments that evaluate the effectiveness of linear and log-linear voting techniques for a variety of sequence prediction tasks. We conclude the chapter with Section 6.8, which discusses the results of these experiments.

6.1 Local Voting vs Global Voting

If we are trying to combine multiple models that all decompose the overall problem in the same way, then we can perform voting on the subproblems, rather than on the overall problem. A common example of this approach is *linear interpolated backoff*, where a subproblem model that is based on a large feature space is smoothed by averaging it with simpler models, based on subsets of the feature space. Linear interpolated backoff can help prevent some types of sparse data problems, by allowing the combined model to fall back on the simpler models' estimates when the more specific model's estimates are unreliable.

However, if the models that we would like to combine all decompose the overall problem in different ways, then voting on subproblem models is not an option. The immediate problem with such an approach is that the subproblems do not correspond to one another. But even if we could align the subproblems, voting on aligned subproblems is still suboptimal: different problem decompositions allow us to encode different long-distance dependencies between output structures; and in order to preserve the information about these dependencies that is contained in the individual models, voting must be done *globally*, on entire output structure, rather than *locally*, on individual subproblems.

Nevertheless, several systems have used local voting schemes. For example, Shen & Sarkar (2005) combines the output of multiple chunkers by converting their outputs to a common format, and taking a majority vote on each tag (Shen and Sarkar, 2005). Since the voting is done on individual elements, and not on sequences, there is no guarantee that the overall sequence will be assigned a high probability by *any* of the individual classifiers. This problem is demonstrated in figure 6.1, which shows the Viterbi graphs generated by three models for a given input. The probability distribution over sequences defined by these graphs is shown in the following table:

Model	**Sequence**	**P(Sequence)**
Model 1	**P-N-V-N**	1.0
Model 2	**P-V-N-N**	0.6
	P-N-V-N	0.4
Model 3	**P-N-N-V**	0.6
	P-N-V-N	0.4

Applying the per-element voting algorithm, we first find the most likely sequence for each model, and then vote on each individual part-of-speech tag. The highest scored sequences are **P-N-V-N** (model 1); **P-V-N-N** (model 2); and **P-N-N-V** (model 3). Voting on individual part-of-speech tags[1] gives a highest score to the sequence **P-N-N-N**. But this sequence is given an overall probability of zero by all three models. If instead we had voted on sequences (using equal weights for the three models), then the combined model would give the following sequence scores: [2]

Model	**Sequence**	**P(Sequence)**
$\widehat{M_+}(\mathbf{y}\vert\mathbf{x})$	**P-N-V-N**	0.6
	P-V-N-N	0.2
	P-N-N-V	0.2
$\widehat{M_\times}(\mathbf{y}\vert\mathbf{x})$	**P-N-V-N**	1.0
	P-V-N-N	0.0
	P-N-N-V	0.0

Thus, we can see that local voting algorithms do not preserve the relationships between adjacent (and non-adjacent) tags that are encoded in the individual models.

[1] (Assuming equal model weights.)

[2] In this simple example, it is possible to exhaustively enumerate the distribution that is generated by the weighted voting technique. However, in most non-toy examples, there are generally an exponential number of sequences with nonzero probability, making it impractical to examine and rank them all.

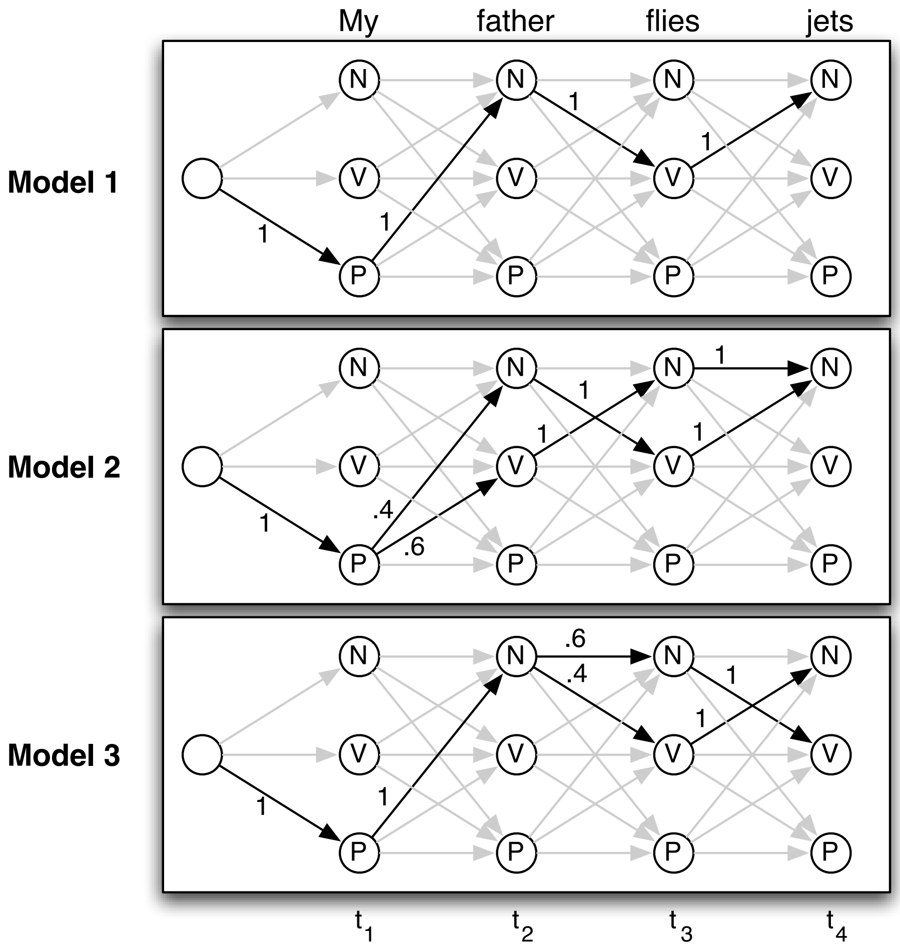


Figure 6.1: **Problematic Set of Models for Local Voting**. The Viterbi graphs generated by three different models for a given input sentence, *"My father flies jets."* (Note that words "father," "flies", and "jets" can each be used as either a verb or noun, depending on context.) The probability for a part-of-speech sequence given by a model is the product of the edges in the path through that sequence of tags (gray edges have probability 0). For example, the probability assigned by model 3 to the sequence **P-N-V-N** is $1 \times 1 \times 0.4 \times 1 = 0.4$; and the probability assigned by model 1 to the sequence **P-N-N-V** is $1 \times 1 \times 0 \times 0 = 0$.

6.2 Global Voting for Sequence-Prediction Tasks

What we would like, then, is to combine the models using a *global* voting method, which determines the structured output value $\mathbf{y}$ with the best voted score. We can do so by applying either linear or log-linear voting to the task of predicting complete structured output values (as opposed to pieces of output values, as was done for local voting):

$$\widehat{M_+}(\mathbf{y}|\mathbf{x}) = \sum_i w_i M_i(\mathbf{y}|\mathbf{x}) \qquad \textit{linear voting} \qquad (6.6)$$

$$\widehat{M_\times}(\mathbf{y}|\mathbf{x}) = \frac{1}{Z(\mathbf{x})} \prod_i M_i(\mathbf{y}|\mathbf{x})^{w_i} \qquad \textit{log-linear voting} \qquad (6.7)$$

For classification tasks, we are able to find the highest-scoring output $\mathbf{y}^*$ by simply calculating $\widehat{M_+}(\mathbf{y}|\mathbf{x})$ or $\widehat{M_\times}(\mathbf{y}|\mathbf{x})$ for each possible class label $\mathbf{y}$. However, for structured prediction problems, there are typically an exponential number of possible output values $\mathbf{y}$. Thus, it is impractical to exhaustively calculate the voted score for every possible output value $\mathbf{y}$.

Instead, we can make use of dynamic programming techniques to find the output value with the highest voted score. However, in doing so, we will face two challenges. First, when performing voting between models that decompose a problem in different ways, we will need to "align" the models in some way to allow their incremental scores to be meaningfully combined by the dynamic programming algorithm. And second, as we will see in Section 6.5.2, prediction for linear voting is NP-hard in the general case. However, we will present an algorithm that uses careful pruning to make it possible to perform prediction for linear voting exactly in the common case, and approximately in the general case.

For the remainder of this Chapter, we will focus on sequence prediction tasks. We assume that each of the individual models that are being combined encodes $P(\mathbf{y}|\mathbf{x})$ using a Viterbi Graph.[3] Based on this assumption, we will describe dynamic

[3] This assumption holds if the individual models are HMMs, MEMMs, CRFs, or other related

programming algorithms that can be used to perform sequence prediction for linear and log-linear voting. Each of these algorithms takes as their input two or more Viterbi Graphs (encoding the individual models' probability distributions); and combines them into a single graph, which is then used to find the highest-scoring output value.

We will begin, in Section 6.3, by describing an algorithm for performing log-linear voting between models that decompose the sequence prediction problem in the same way. In Section 6.4, we will show how that algorithm can be generalized to work for models that decompose the sequence prediction problem in different ways. We will then turn to the problem of linear voting. Again, we will start in Section 6.5 with the simpler case of combining models that use the same problem decomposition; and then, in Section 6.6, we will generalize that algorithm to work for models with differing problem decompositions.

6.3 Log-Linear Voting for Models with Identical Problem Decompositions

In (Smith et al., 2005), Smith et al. discuss the use of log-linear voting to combine multiple CRF-based sequence models that use the same output encoding. They refer to this system-combination technique as "logarithmic opinion pools" (or "LOP"), since it combines multiple models (pooling) using log-linear voting. In (Smith et al., 2005; Smith and Osborne, 2007), they show how this technique can be used to perform voting between models that use different feature sets, or that are trained on different corpora.

The prediction algorithm for log-linear voting when the models use the same encoding is fairly straight forward. First, we construct a combined Viterbi graph, whose edge weights are a weighted product of the edge weights of the individual

models.

models' Viterbi graphs. I.e., letting $v_s^{M_i}(1)$ and $v_{ss'}^{M_i}(t)$ be the Viterbi graph weights for model M_i, we define a new Viterbi graph (with the same states and edges as the graphs we're combining) whose weights $v_s^{\widehat{M}}(1)$ and $v_{ss'}^{\widehat{M}}(t)$ are defined as:

$$v_s^{\widehat{M}}(1) = \frac{1}{Z(\mathbf{x})} \prod_i \left(v_s^{M_i}(1)\right)^{w_i} \tag{6.8}$$

$$v_{ss'}^{\widehat{M}}(t) = \prod_i \left(v_{ss'}^{M_i}(t)\right)^{w_i} \tag{6.9}$$

The score assigned to any path by this combined Viterbi graph will be equal to the weighted product of the scores assigned by the individual Viterbi graphs. We can therefore use standard Viterbi decoding on the combined Viterbi graph to find the output sequence with the highest voted score.

To see why this method of combining the Viterbi graphs works, recall that the log-linear voting model is defined as:

$$\widehat{M_\times}(\mathbf{y}|\mathbf{x}) = \frac{1}{Z(\mathbf{x})} \prod_i M_i(\mathbf{y}|\mathbf{x})^{w_i} \tag{6.10}$$

Substituting in the Viterbi probability decomposition (Equation 2.15) for each model $M_i(\mathbf{y}|\mathbf{x})$ gives:

$$\begin{aligned}
\widehat{M_\times}(\mathbf{y}|\mathbf{x}) &= \frac{1}{Z(\mathbf{x})} \prod_i \left(v_{y_1}^{M_i}(1) \prod_{t=2}^{T} v_{y_{t-1}y_t}(t) \right)^{w_i} && (6.11)\\
&= \frac{1}{Z(\mathbf{x})} \prod_i \left(v_{y_1}^{M_i}(1)^{w_i} \prod_{t=2}^{T} v_{y_{t-1}y_t}(t)^{w_i} \right) && (6.12)\\
&= \frac{1}{Z(\mathbf{x})} \prod_i v_{y_1}^{M_i}(1)^{w_i} \prod_i \prod_{t=2}^{T} v_{y_{t-1}y_t}(t)^{w_i} && (6.13)
\end{aligned}$$

Since multiplication is commutative, we can transpose the two products:

$$\widehat{M_\times}(\mathbf{y}|\mathbf{x}) = \frac{1}{Z(\mathbf{x})} \prod_i v_{y_1}^{M_i}(1)^{w_i} \prod_{t=2}^{T} \prod_i v_{y_{t-1}y_t}(t)^{w_i} \tag{6.14}$$

Substituting in the definitions of $v_s^{\widehat{M}}(1)$ and $v_{ss'}^{\widehat{M}}(t)$ from Equations 6.8 and 6.9, we are left with the same equation that is used to calculate the probabilities of paths in a Viterbi graph:

$$\widehat{M_\times}(\mathbf{y}|\mathbf{x}) = v^{\widehat{M}}_{y_1}(1)\prod_{t=2}^{T} v^{\widehat{M}}_{y_{t-1}y_t}(t) \tag{6.15}$$

6.4 Log-Linear Voting for Models with Differing Problem Decompositions

Unfortunately, this technique can not be used to combine models that use different problem decompositions, because their Viterbi graphs are incompatible. In particular, their Viterbi graphs will use a different set of states (or use the same states, but with different meanings). However, if we can "align" the two graphs, then it will be possible to combine them using a method similar to the method described in Section 6.3. Following on the work in Chapter 4, I will assume that each model's encoding is represented concretely as a Finite State Transducer that maps from canonical tag strings to encoded tag strings.

6.4.1 A Simple Example Case

To see how this might work, we will first consider a simple example case, shown in Figure 6.2. In this example, we wish to combine two models for a sequence labeling task whose canonical encodings use the tags A and B. The first model has subdivided the A tag into two sub-tags, in order to record information about the previous state. Similarly, the second model has subdivided the B tag into two sub-tags.

As a first step towards finding the output value with the highest voted score, we will construct a combined Viterbi graph that can be used to find the weighted product of the scores of *any* pairing of paths through the two individual models' Viterbi graphs. The states of this combined Viterbi graph are tuples encoding all possible pairings of states from the individual models' Viterbi graphs:

$$S^{\widehat{M}} = \left\{ \langle s_1, s_2 \rangle : s_1 \in S^{M_1}; s_2 \in S^{M_2} \right\} \tag{6.16}$$

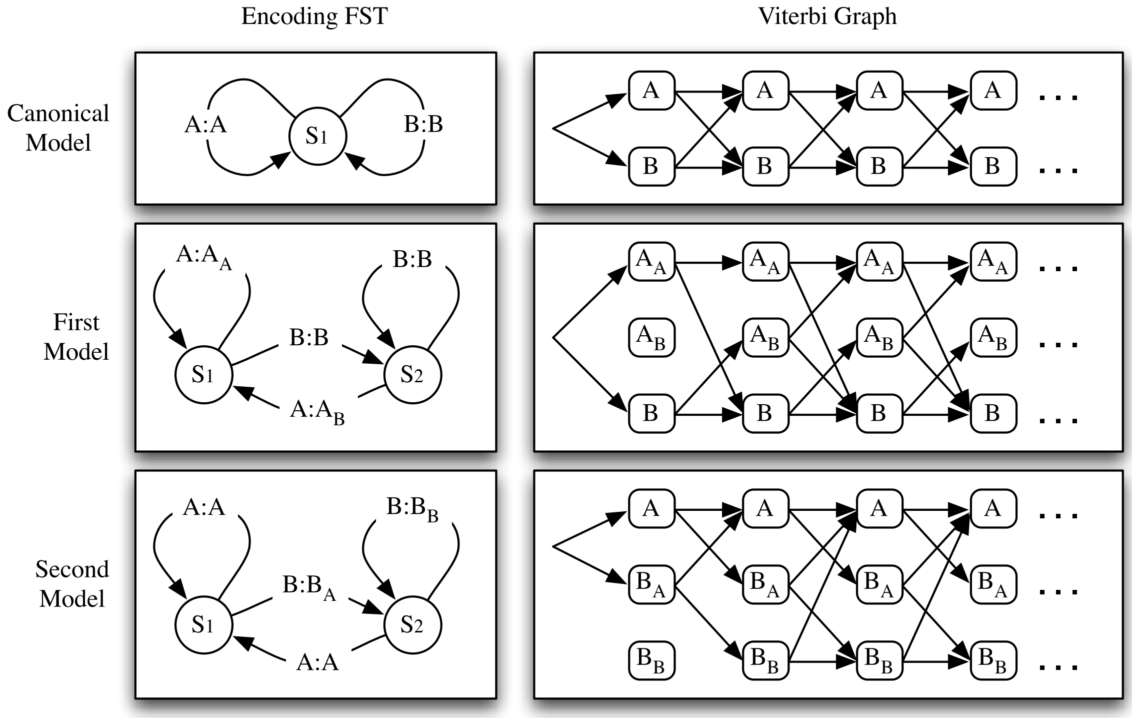


Figure 6.2: **Example Case for Log-Linear Voting with Differing Problem Decompositions**. In this example, we wish to combine the Viterbi graphs generated by two models ("first model" and "second model") using log-linear voting. However, these two models use different problem decompositions, so their Viterbi graphs can not be directly combined using the method described in Section 6.3.

The edge weights for this new combined graph are simply the weighted products of the edge weights for the individual models' Viterbi graphs:

$$v^{\widehat{M}}_{\langle s_1, s_2 \rangle}(1) = v^{M_1}_{s_1}(1)^{w_1} v^{M_2}_{s_2}(1)^{w_2} \tag{6.17}$$

$$v^{\widehat{M}}_{\langle s_1, s_2 \rangle \langle s_1', s_2' \rangle}(t) = v^{M_1}_{s_1 s_1'}(t)^{w_1} v^{M_2}_{s_2 s_2'}(t)^{w_2} \tag{6.18}$$

Given these definitions, we can see that the score assigned to any path by the combined Viterbi graph is equal to the weighted product of the scores assigned to the corresponding paths by the individual model's Viterbi graphs:

$$\mathit{score}^{\widehat{M}}(\langle y_1^{M_1}, y_1^{M_2} \rangle, ... \langle y_T^{M_1}, y_T^{M_2} \rangle) \tag{6.19}$$

$$= v^{\widehat{M}}_{\langle y_1^{M_1}, y_1^{M_2} \rangle}(1) \prod_{t=2}^{T} v^{\widehat{M}}_{\langle y_{t-1}^{M_1}, y_{t-1}^{M_2} \rangle \langle y_t^{M_1}, y_t^{M_2} \rangle}(t) \tag{6.20}$$

$$= v^{M_1}_{y_1^{M_1}}(1)^{w_1} v^{M_2}_{y_1^{M_2}}(1)^{w_2} \prod_{t=2}^{T} v^{M_1}_{y_{t-1}^{M_1} y_t^{M_1}}(t)^{w_1} v^{M_2}_{y_{t-1}^{M_2} y_t^{M_2}}(t)^{w_2} \tag{6.21}$$

$$= \left(v^{M_1}_{y_1^{M_1}}(1) \prod_{t=2}^{T} v^{M_1}_{y_{t-1}^{M_1} y_t^{M_1}}(t) \right)^{w_1} \left(v^{M_2}_{y_1^{M_2}}(1) \prod_{t=2}^{T} v^{M_2}_{y_{t-1}^{M_2} y_t^{M_2}}(t) \right)^{w_2} \tag{6.22}$$

$$= \mathit{score}^{M_1}(y_1^{M_1}, ..., y_T^{M_1})^{w_2} \mathit{score}^{M_2}(y_1^{M_2}, ..., y_T^{M_2})^{w_2} \tag{6.23}$$

Thus, the combined Viterbi graph contains a single path corresponding to each possible output value, which is formed by combining the individual models' encodings of that output value, and whose score is equal to the voted score for that output value. However, as we can see in Figure 6.3, the combined Viterbi graph also contains paths that do not correspond to any single output value. The basic problem here is that the combined graph includes paths for "inconsistent" pairings of paths the two individual models – i.e., pairings where the two paths describe different output values.

In the case of our example problem, the solution is quite simple: we just discard any combined states that are inconsistent, such as $\langle A_A, B_A \rangle$. The resulting graph, shown in Figure 6.4, only contains paths that pair together equal output values. It

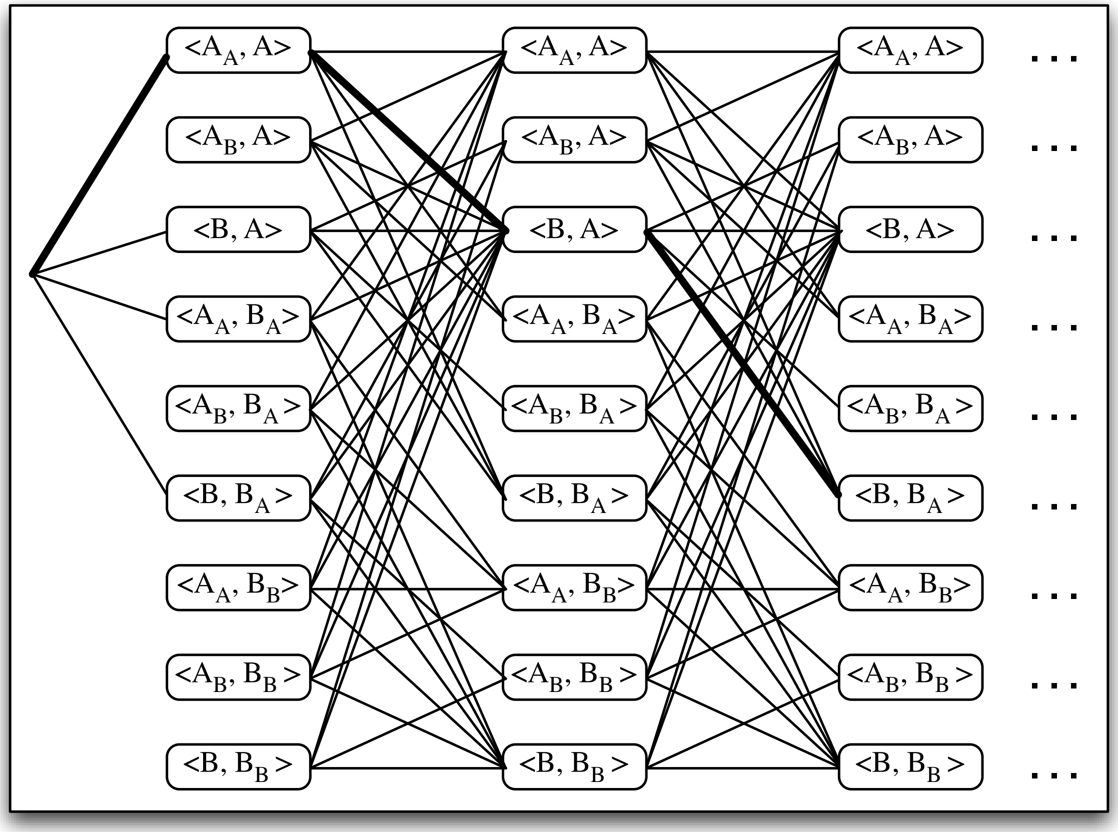


Figure 6.3: **Combined Viterbi Graph for the Log-Linear Voting Example**. Each state in the combined Viterbi graph is a tuple $\langle s_1, s_2\rangle$, where $s_1 \in S^{M_1}$ and $s_2 \in S^{M_2}$. The weight of each edge in this combined Viterbi graph is the weighted product of the corresponding edges for the two model's Viterbi graphs. The (partial) path shown in bold is an example of a path that does not correspond to a single output value. In particular, if we take the first element of each state tuple, we get the sequence $(A_A, B, B, ...)$; but if we take the second element of each state tuple we get the sequence $(A, A, B_A, ...)$. These two sequences describe different output values.

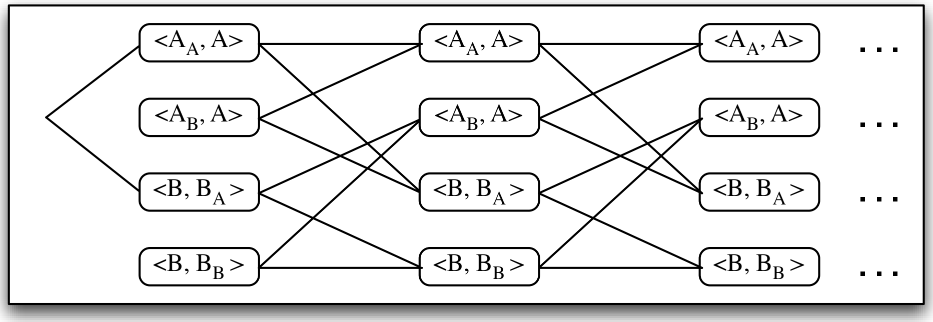


Figure 6.4: **Pruned Viterbi Graph for the Log-Linear Voting Example**. The graph that results from removing any inconsistent states from the combined Viterbi graph shown in Figure 6.3. By removing the inconsistent states, we ensure that the resulting graph only contains paths that pair together equal output values.

can therefore be used to find highest-scoring output value by using the standard Viterbi decoding algorithm.

In this example case, it was fairly straight-forward to align the individual models' output labels because they were simply sub-labels of the canonical tag set. However, in the general case, the relationship between two encodings may be more complex. But as we will see, it is possible to align their Viterbi graphs by making use of the FSTs that define their output encoding.

Before we describe how to align the Viterbi graphs for a collection of models that use arbitrary FSTs to generate their encoded values, we will introduce a new formalism: the Grouped-State Viterbi Graph. This formalism will also be useful when we consider the case of linear voting.

6.4.2 Grouped-State Viterbi Graphs

A Grouped-State Viterbi Graph (or GSVG) is simply a Viterbi graph $\langle S, T, Q, E\rangle$ that is augmented with a set of groups G and a grouping function $g(s)$ that maps each state $s \in S$ to a group $g \in G$. Graphically, we depicted grouped-state viterbi graphs by drawing a circle around the graph nodes whose states are in the same group. Two examples of Grouped State Viterbi Graphs (corresponding to the example from the

Grouped-State Viterbi Graph	$\langle S, T, Q, E, G, g(s)\rangle$		
Graph Nodes	Q	$=$	$\{q_0\} \cup \{q_{t,s} : 1 \leq t \leq T; s \in S\}$
Graph Edges	E	$=$	$\{\langle q_0 \rightarrow q_{1,s}\rangle : s \in S\} \cup$
			$\{\langle q_{t-1,s} \rightarrow q_{t,s'}\rangle : s \in S; t \in T\}$
Graph State Groups	G	$=$	$\{g_1, ..., g_K\}$
Grouping Function	$G(s)$	$:$	$S \rightarrow G$
State Sequence	$\vec{y}$	$=$	$(y_1, ..., y_T)$
State Group Sequence	$\vec{r}$	$=$	$(r_1, ..., r_T)$

Figure 6.5: **Notation for Grouped-State Viterbi Graphs**.

previous Section) are shown in Figure 6.6.

6.4.3 Log-Linear Voting with Grouped-State Viterbi Graphs

For log-linear voting, we will translate each model's Viterbi graph into a GSVG, where the groups are used to indicate which states are compatible with one another. In particular, we can perform log-linear voting by using the following basic algorithm:

1. Compute the Viterbi graph for each model.

2. Use each model's FST to transform its Viterbi graph into a Grouped State Viterbi Graph, whose groups correspond to the tags used by the canonical representation.

3. Combine these Grouped State Viterbi Graphs into a merged GSVG, whose states pair together states from the individual GSVGs that belong to the same group.

4. Use the standard Viterbi decoding algorithm to find the highest scoring path through the combined GSVG.

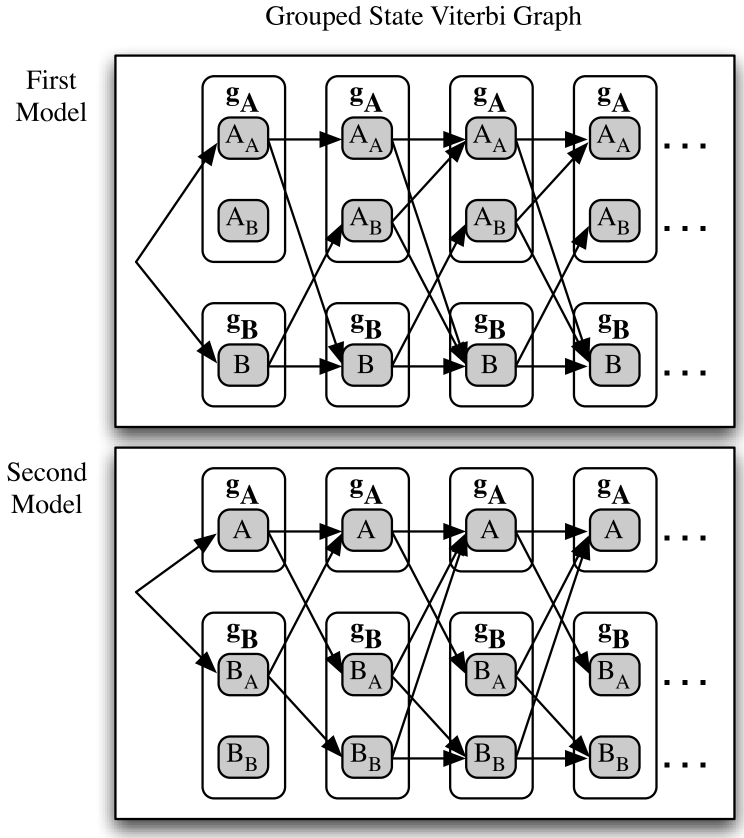


Figure 6.6: **Grouped State Viterbi Graphs for the Log-Linear Voting Example**. This example shows how the two Viterbi graphs shown in Figure 6.2 could be “aligned” by converting them to GSVGs. Any pair of paths from the two GSVGs are considered compatible iff they pass through the same set of groups. This ensures that they both describe the same output value.

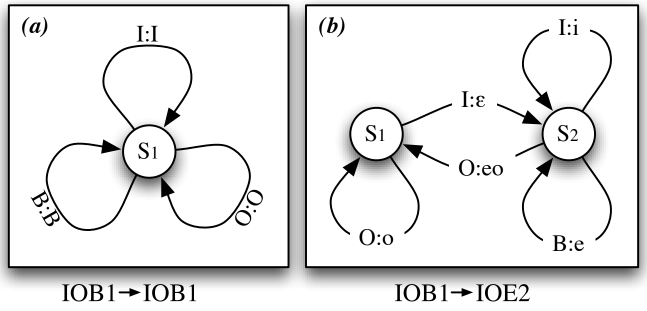


Figure 6.7: **FSTs Representing the `IOB1` and `IOE2` Encodings**. We will use these two encodings as an example to illustrate the global voting algorithm with multiple encodings. Both FSTs are expressed with respect to the canonical encoding `IOB1`. See Section 2.2.2 for an explanation of the IOB1 and IOE2 encodings. These two encodings are also summarized in Figure 6.17.

As a running example, I will consider the voted combination of two systems: one using the `IOB1` encoding, and the other using the `IOE2` encoding.[4] The FST representations for these two encodings are shown in Figure 6.7. In order to help distinguish the "`I`" and "`O`" tags used by `IOE2` from those used by `IOB1`, I will use the lower case letters "`i`" "`o`" and "`e`" to denote the `IOE2` tags; and the upper case letters "`I`" "`O`" and "`B`" to denote the `IOB1` tags. The Viterbi graphs for these two models are shown in Figure 6.8.

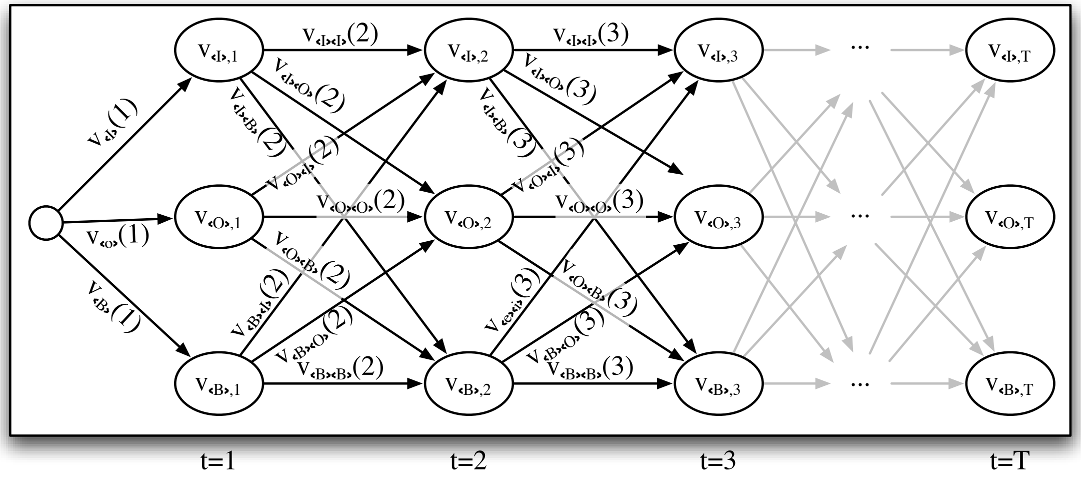
Transforming Viterbi Graphs into Canonical-Encoding GSVGs

Before we can perform log-linear voting between models that use different encodings, we must first "align" their Viterbi graphs. We do this by constructing a GSVG for each model's Viterbi graph whose groups correspond to the tags used by the canonical representation. This GSVG, which we will call the "Canonical-Encoding GSVG" for a given model, needs to satisfy the following properties:

1. Each path p through the graph's subnodes corresponds to exactly one structured output value $value(p)$.

[4] I will take `IOB1` to be the canonical encoding. As discussed in Chapter 4, the choice of canonical encoding does not have an effect on the expressive power of FSTs as a means of expressing encodings.

IOB



IOE

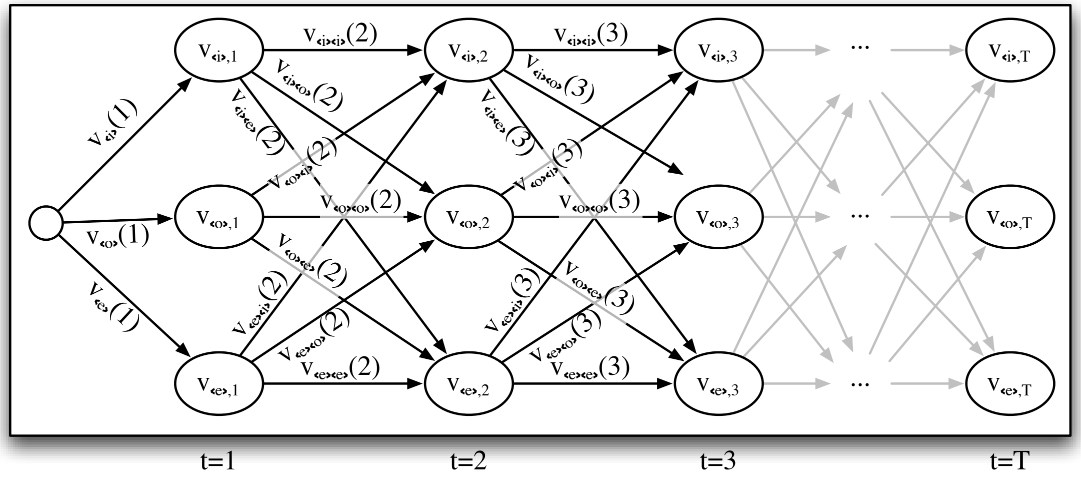


Figure 6.8: **Viterbi Graphs for the `IOB1` and `IOE2` Encodings**. These two Viterbi graphs can not be directly combined, because they use incompatible states: the `IOB1` graph uses the three states `I`, `O`, and `B`, while the `IOE2` graph uses the states `i`, `o`, and `e`. We must therefore transform the graphs into a common format before they can merged.

2. The score for path p is equal to the score assigned by the original Viterbi graph to $value(p)$.

3. For each path $p = (y_1, ..., y_T)$, the corresponding value $value(p)$ is encoded (in the canonical encoding) by the sequence $(g(y_1), ..., g(y_T))$.

Property (2) ensures that the new canonicalized graph encodes the same probability distribution as the original Viterbi Graph. Property (3) will allow us to combine this canonicalized graph with other model's canonicalized graphs, by simply merging the corresponding group nodes.

In order to construct the Canonical-Encoding GSVG, we will make use of the FST representing the model's encoding. Recall that this FST translates from tag sequences encoded with the canonical encoding to tag sequences encoded with the model's encoding. We will begin by normalizing this FST such that every edge contains exactly one input symbol. Since the FST's input symbols are tags in the canonical encoding, this means that each time we step through the FST, we will consume exactly one canonical tag, and generate zero or more encoded tags.

We will represent each output value in the Canonical-Encoding GSVG by modelling the path that is taken through the encoding FST for that value. In particular, for each node in the path through the FST, the Canonical-Encoding GSVG will contain a corresponding subnode; and the score of the path through these subnodes will equal the score of the encoded value.

Each subnode in the Canonical-Encoding GSVG must contain enough information about the process of running the FST that we can calculate appropriate edge scores between subnodes. In particular, each subnode must keep track of:

- The current state of the FST.

- The most recently consumed canonical tag. This determines which group the node will belong to.

- The most recently generated encoded tag. This determines which of the scores from the original Viterbi Graph we will use to compute the score of outgoing edges from this subnode in the new Canonical-Encoding GSVG.

- The t index of the most recently consumed canonical tag. This determines in which time slice the node will be located in the new Canonical-Encoding GSVG.

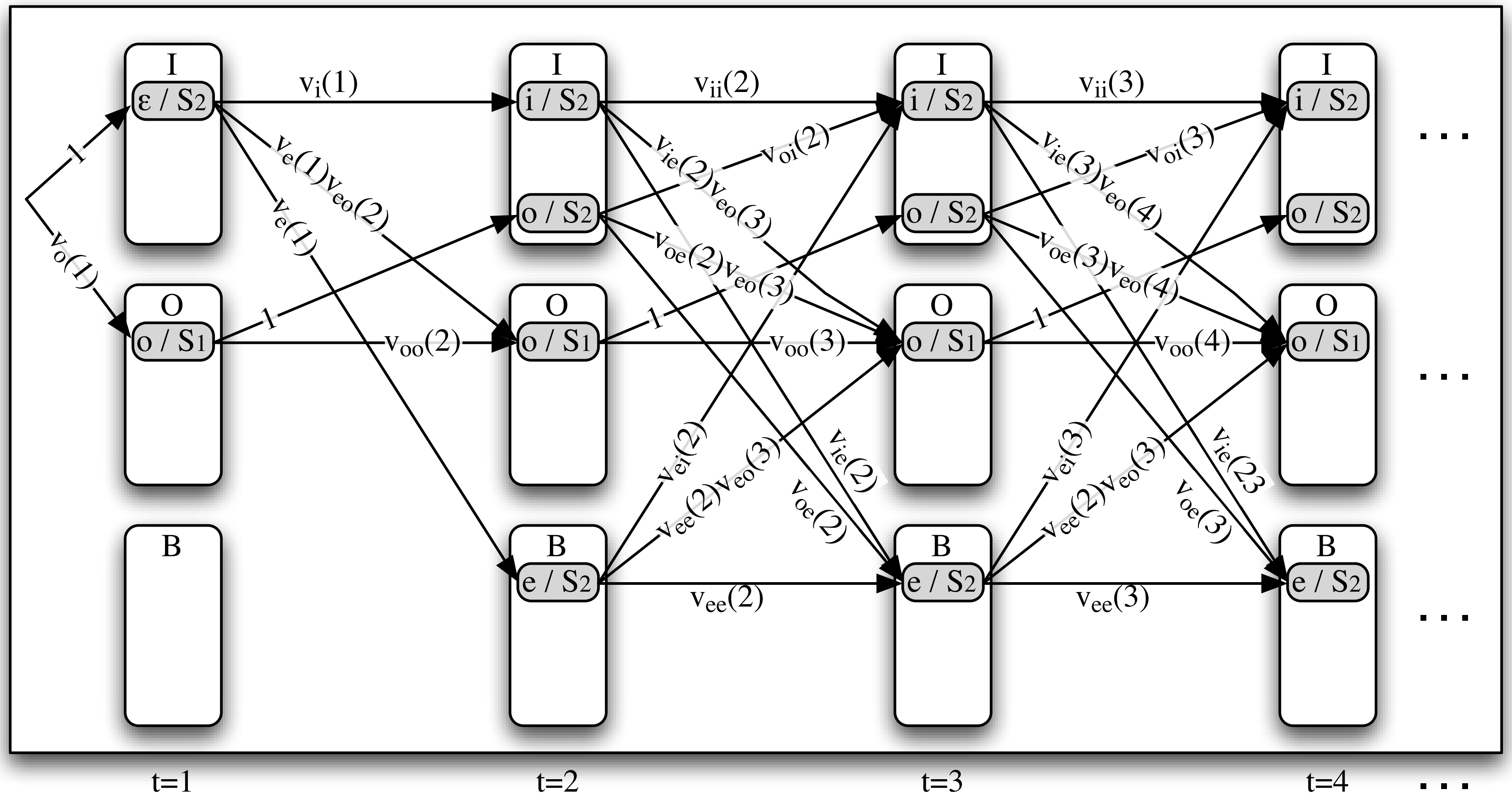


Figure 6.9: **Canonical-Encoding GSVG for an `IOE2` model**. The Canonical-Encoding GSVG contains one grouped state for each tag in the canonical encoding, allowing it to be combined with other models' Canonical-Encoding GSVGs. Each node in the graph is a tuple $\langle s, t, tag_c, tag_e \rangle$, where s is a state in the encoding FST; t is a time value; tag_c is a canonical tag; and tag_e is an encoded tag. In the figure, these nodes are simply labeled as "tag_e/s"; the t value is implicit in the node's column, and the tag_c value is implicit in the node's group.

Thus, we will define each subnode to be a tuple $\langle s, t, tag_c, tag_e \rangle$, where s is a state in the encoding FST; t is a time value; tag_c is a canonical tag; and tag_e is an encoded tag. We do not need to include all possible subnodes; instead, we can determine which subnodes will be used by exploring the paths that the FST can take. The score of an edge in the new Canonical-Encoding GSVG is computed by examining the sequence of encoded tags generated by the the FST during the corresponding step; and multiplying their score in the original Viterbi graph. This algorithm is shown in detail in Figure 6.10; and the resulting Canonical-Encoding GSVG for an `IOE2` model is shown in Figure 6.9.

Combining Canonical-Encoding GSVGs

Once each model's Viterbi Graph has been converted to a Canonical-Encoding GSVG, we can combine those individual GSVGs into a single GSVG that simulates the process of simultaneously tracing paths through each of the individual GSVG. This combined GSVG's states will be tuples encoding possible pairings of states from the individual GSVGs, with the restriction that the states must all belong to the same group:

$$S^{\widehat{M}} = \Big\{ \langle s_1, s_2, ..., s_N \rangle : s_i \in S^{M_i}; \forall i \forall j g(s_i) = g(s_j) \Big\} \tag{6.24}$$

The edge weights for the combined GSVG are simply the weighted products of the edge weights for the individual models' Canonical-Encoding GSVG:

$$v^{\widehat{M}}_{\langle s_1, ..., s_N \rangle}(1) = \prod_i v^{M_i}_{s_i}(1)^{w_i} \tag{6.25}$$

$$v^{\widehat{M}}_{\langle s_1, ..., s_N \rangle \langle s'_1, ..., s'_N \rangle}(t) = \prod_i v^{M_i}_{s_i s'_i}(t)^{w_i} \tag{6.26}$$

Once this merged GSVG has been created, we can find the highest-scoring output value by applying the standard Viterbi decoding algorithm to find the highest scoring path $y^* = (y_1, ..., y_T)$; and then converting that path to a canonically-encoded value r^* by checking which group each y_i belongs to:

```
def canonicalize_viterbi_graph(graph, fst):
    # Canonicalize the FST such that each edge's input string contains
    # exactly one symbol.  Output strings may be empty, or may contain
    # multiple characters.
    fst.canonicalize(input_len=1)

    # Create the new Grouped State Viterbi Graph.  Seed it with an
    # initial start node at time 0.
    gsvg = GroupedStateViterbiGraph()
    initial_node = Node(state=fst.initial_state, t=0,
                        c_tag=START, e_tag=START)
    gsvg.add_node(initial_node, group=START)

    # For each time step, examine all nodes at that time step.  Each
    # node corresponds to a possible state of the FST as we convert an
    # output value.
    for t in range(1, graph.T):
        for node in gsvg.nodes(t=t):

            for fst_edge in fst.outgoing_edges(node.state):
                score = 1
                e_tag = node.e_tag
                offset = node.state.outputoffset
                for e_tag2 in fst_edge.output:
                    offset -= 1
                    score *= graph.score(e_tag, e_tag2, t-offset)
                    e_tag = e_tag2

                new_node = Node(state=fst_edge.dest, t=t,
                                c_tag=fst_edge.input, e_tag=e_tag)
                gsvg.add_node(new_node, group=fst_edge.input)
                gsvg.add_edge(node, new_node, score)

    # Return the complete graph.
    return gsvg
```

Figure 6.10: **Canonical Grouped State Viterbi Graph Construction Algorithm**.

1. Compute the Viterbi graph for each model.
2. Convert each model's Viterbi graph into a Canonical-Encoding GSVG, using the algorithm shown in Figure 6.10.
3. Combine these Grouped State Viterbi Graphs into a merged GSVG, as described in Section 6.4.3.
4. Use the standard Viterbi decoding algorithm to find the highest scoring path through the combined GSVG.

Figure 6.11: **Summary of the Sequence Prediction Algorithm for Log-Linear Voting**.

$$r^* = (g(y_1), g(y_2), ..., g(y_T)) \tag{6.27}$$

Figure 6.11 gives a summary of the overall algorithm used to perform sequence prediction for log-linear voting.

6.4.4 Log-Linear Voting with FST Composition

It is also possible to perform log-linear voting using a different but related algorithm, based on FST composition. The basic idea of this algorithm is to use composition to combine the FST for each model's encoding with that model's Viterbi graph. The resulting composed FSTs map directly from canonically encoded output strings to score sequences. These FSTs have the same basic structure as a Viterbi graph, and are all defined on the same input space (namely, canonically encoded output values). They can therefore be combined using the algorithm presented in Section 6.3.

To understand how this algorithm works, first note that Viterbi Graphs can be thought of as finite state transducers that map from encoded output values to sequences of weights. When treated as a transducer, the score for a path through the Viterbi Graph is found by first mapping the encoded output value to a sequence of scores; and then taking their product.

Since Viterbi Graphs can be thought of as FSTs, they can be combined with other FSTs using composition. In particular, let $\textsc{Enc}^{M_i}$ be the encoding transducer for model M_i; and let $\textsc{Viterbi}^{M_i}$ be the Viterbi Graph generated by model M_i for a given input. Note that $\textsc{Enc}^{M_i}$ maps the canonical encoding to the model's encoding; and that $\textsc{Viterbi}^{M_i}$ maps from the model's encoding to score sequences. The composed transducer, $\textsc{Enc}^{M_i} \cdot \textsc{Viterbi}^{M_i}$, therefore maps from canonical encodings to score sequences. This composed transducer is created using the standard algorithm for composing FSTs. In particular:

- The composed FST will have one state for each pair of states (e_i, v_i) where e_i is a state from $\textsc{Enc}^{M_i}$ and v_i is a state from $\textsc{Viterbi}^{M_i}$.
- The composed FST will have an edge $\langle (e_i, v_i) \rightarrow (e_j, v_j)[a : s] \rangle$ iff both:
  - $\textsc{Enc}^{M_i}$ contains the edge $\langle e_i \rightarrow e_j[a : b] \rangle$; and
  - $\textsc{Viterbi}^{M_i}$ contains the edge $\langle v_i \rightarrow v_j[b : s] \rangle$

If we assume that $\textsc{Enc}^{M_i}$ is normalized so that each edge consumes a single canonical output tag, then the resulting transducer will also have edges that consume a single output tag. It is therefore structurally identical to the FSTs that we use to encode Viterbi Graphs, and the standard Viterbi Graph algorithms can be applied to it. In particular, once we have composed each model's encoding transducer with its Viterbi Graph, we can then perform voting between those models by simply applying the algorithm presented in Section 6.3.

6.5 Linear Voting for Models with Identical Problem Decompositions

We will now turn our attention to linear voting methods, which use a weighted sum (rather than a weighted product) to calculate the voted score for a given output value. We will begin by examining the simpler problem of performing global voting for a set of models that all use the same problem decomposition. We will then show how these techniques can be applied to sets of models that use different problem decompositions.

Using linear voting, the score that is assigned to any output value $\vec{y}$ is the weighted sum of the scores assigned to that value by the individual models:

$$\widehat{M_+}(\mathbf{y}|\mathbf{x}) = \sum_i w_i M_i(\mathbf{y}|\mathbf{x}) \tag{6.28}$$

Substituting in the Viterbi probability decomposition for each model $M_i(\mathbf{y}|\mathbf{x})$ gives:

$$\widehat{M_+}(\mathbf{y}|\mathbf{x}) = \sum_i w_i v_{y_1}^{M_i}(1) \prod_{t=2}^{T} v_{y_{t-1}y_t}^{M_i}(t) \tag{6.29}$$

The highest-scoring output value $\mathbf{y}^*$ is therefore given by:

$$\mathbf{y}^* = \arg\max_{\mathbf{y}} \sum_i w_i v_{y_1}^{M_i}(1) \prod_{t=2}^{T} v_{y_{t-1}y_t}^{M_i}(t) \tag{6.30}$$

Unfortunately, Equation 6.30 can not be solved using dynamic programming techniques. The presence of an extra summation between the arg max and the product prevents us from recursively calculating $\delta_s(t)$, the score of the highest scoring path through the Viterbi graph node $q_{t,s}$. We therefore cannot apply the Viterbi algorithm. Note that this contrasts with the case of log-linear voting, where the corresponding equation had two products rather than a product and a sum; and thus we were able to apply the Viterbi algorithm by transposing the two products.

One interpretation of equation 6.30 is that we are looking for a single path, specified by the tag sequence $\vec{y}^*$, that maximizes the total score generated by a set of Viterbi matrices, $\left\{v^{M_i}\right\}$. This interpretation can be made concrete by combining the individual models' Viterbi graphs into a single Grouped-State Viterbi Graph, where the corresponding nodes from each graph are combined together in a single group; and reformulating our goal as finding the highest-scoring sequence of *groups*.

In particular, we will define the score of a group sequence $\vec{r} = (r_1, ..., r_T)$ to be the sum of the scores of all state sequences $\vec{y} = (y_1, ..., y_T)$ that are consistent with it:

$$\begin{aligned}\text{score}(\vec{r}) &= \sum_{\vec{y}:\forall t, g(y_t)=r_t} \text{score}(\vec{y}) && (6.31)\\ &= \sum_{\vec{y}:\forall t, g(y_t)=r_t} \left(v_{y_1}(1) \prod_{t=2}^{T} v_{y_{t-1}y_t}(t) \right) && (6.32)\end{aligned}$$

In order to use Grouped-State Viterbi Graphs to perform global voting for a sequence-prediction task, we must first combine the individual models' Viterbi Graphs into a single Grouped-State Viterbi Graph. We construct this combined graph as follows:

1. States in the new Grouped-State Viterbi Graph will be tuples $\langle M_i, s_j \rangle$, pairing a model with one of that model's states.

2. State groups will consist of the corresponding states from each model:

$$\begin{aligned} G &= S && (6.33)\\ g(\langle M_i, s \rangle) &= s && (6.34)\end{aligned}$$

3. The Grouped-State Viterbi Graph transition scores will simply be copied from the individual models' Viterbi Graphs, with the initial edge weights modified to account for the model weights w_i:

$$v^{\widehat{M}}_{\langle M_i, s \rangle}(1) = w_i v^{M_i}_s(1) \qquad (6.35)$$

$$v^{\widehat{M}}_{\langle M_i, s \rangle \langle M_i, s' \rangle}(t) = \left(v^{M_i}_{ss'}(t) \right) \qquad (6.36)$$

$$v^{\widehat{M}}_{\langle M_i, s \rangle \langle M_j, s' \rangle}(t) = 0 \qquad (\forall i \neq j) \qquad (6.37)$$

In essence, this new Grouped-State Viterbi Graph simply combines the individual Viterbi graphs by grouping the corresponding nodes. Given this Grouped-State Viterbi Graph, and the definition of score($\vec{r}$) given in Equation 6.31, the score of a group sequence will be equal to the weighted average of the scores given by the individual models to the corresponding state sequence. Thus, finding the highest scoring voted sequence is equivalent to finding the highest scoring group sequence through the Grouped-State Viterbi Graph:

$$\begin{aligned} \vec{y}^* &= \arg\max_{\vec{y}} \widehat{M}(\vec{y}|\vec{x}) && (6.38) \\ &= \arg\max_{\vec{y}} \sum_i w_i \text{score}^{(M_i)}(\vec{y}) && (6.39) \\ &= \arg\max_{\vec{r}} (\text{score}(\vec{r})) && (6.40) \end{aligned}$$

Unfortunately, the problem of finding the optimal group sequence in a Grouped-State Viterbi Graph is NP-hard in the general case (See Appendix A for a proof.). But the following sections will present algorithms that can be used to find the optimal G in common cases; or to find a near-optimal G in all cases.

6.5.1 Finding Optimal Group Sequences

Although the problem of finding the optimal path through a Grouped-State Viterbi Graph is NP-Hard in the general case, it is still possible to derive algorithms which can find the optimal path for a restricted set of graphs; or to find a near-optimal path for any graph. In this section, we will first develop an algorithm that can be used to find the optimal group sequence for most of the Grouped State Viterbi Graphs that are generated by common machine learning algorithms. We will then show how this algorithm can be made to cover all inputs, at the expense of producing a near-optimal path.

6.5.2 Why Dynamic Programming Fails

Recall that for a simple Viterbi graph, we found the optimal path using dynamic programming. We would like to use a similar approach for the more general case of Grouped-State Viterbi Graphs. For simple Viterbi graphs, we defined a variable $\delta_s(t)$, which recorded the score of the highest scoring path from the start node q_0 to

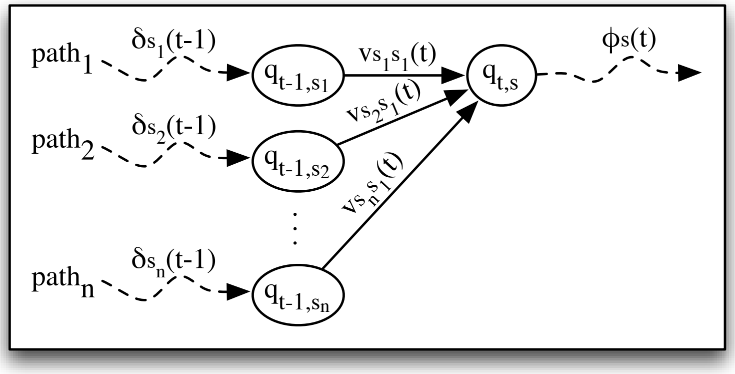


Figure 6.12: **Viterbi Dynamic Programming Decomposition.** A portion of a Viterbi graph, showing the score decomposition that is used by the Viterbi algorithm to calculate $\delta_s(t)$ recursively.

the node $q_{t,s}$. This variable can be calculated recursively; and can then be used to find the best overall state sequence.

Figure 6.12 illustrates how $\delta_s(t)$ can be calculated recursively. The highest scoring path through node $q_{t,s}$ must pass through node $q_{t-1,s'}$, for some $s' \in S$. If we can determine which of these source nodes is part of the highest scoring path, then we can simply calculate $\delta_s(t)$ as $\delta_{s^*}(t-1)v_{s^*s}(t)$. In order to determine which of the source nodes is part of the highest scoring path, note that:

$$\begin{aligned}
\arg\max_{path_i}\left(\text{score}(path_i)\right) &= \arg\max_{path_i} \delta_{s_i}(t-1)v_{s_is}(t)\phi_s(t) && (6.41)\\
&= \arg\max_{path_i} \delta_{s_i}(t-1)v_{s_is}(t) && (6.42)
\end{aligned}$$

Thus, we can determine the best path without knowing the max backward score $\phi_s(t)$, since the value of $\phi_s(t)$ does not depend on the choice of $path_i$. This allows us to recursively calculate $\delta_s(t)$ given only v and $\delta_{s_i}(t-1)$.

But in Grouped-State Viterbi Graphs, we are *not* able to determine the best path without knowing the max backward scores $\phi_s(t)$. To understand why, compare Figure 6.12 to Figure 6.13, which shows a comparable portion of a Grouped-State Viterbi Graph. As with simple Viterbi Graphs, the best group-node path must pass through group-node $g_{t-1,i}$ for some group i. So we are interested in determining

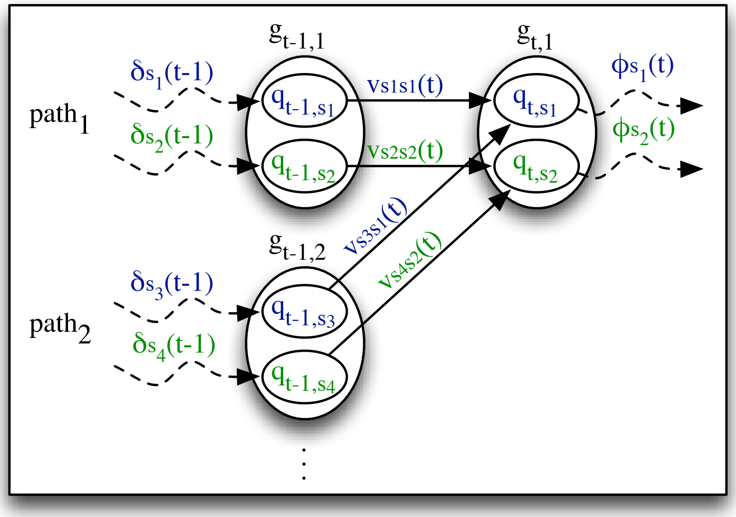


Figure 6.13: **Choosing the Best Path in a Grouped State Viterbi Graph**. This diagram shows a portion of a Grouped State Viterbi Graph, illustrating the difficulty in choosing the best incoming path. This diagram corresponds to the portion of a Viterbi Graph shown in figure 6.12.

which of these group-node paths has the highest score:

$$\arg\max_{path_i} (\text{score}(path_i)) = \arg\max_{path_i} \sum_{s' \in g_{t-1,i}} \sum_{s \in g_{t,1}} \delta'_s(t-1) v_{s's}(t) \phi_s(t) \quad (6.43)$$

But there is crucial difference between Equation 6.41 and Equation 6.43: $\phi_s(t)$ is no longer a constant value, so we can not drop it from the arg max.

An Example

An example will help illustrate the problem. Consider the case where we are performing global voting between two models, illustrated in Figure 6.13. The paths that pass through node q_{t,s_1} correspond to model M_1, while the paths that pass through node q_{t,s_2} correspond to model M_2. The overall score that we are trying to maximize will be the sum of one score from each model:

		From M_1		From M_2
score($path_1$)	$=$	$\delta_{s_1}(t-1)v_{s_1s_1}(t)\phi_{s_1}(t)$	$+$	$\delta_{s_2}(t-1)v_{s_2s_2}(t)\phi_{s_2}(t)$
score($path_2$)	$=$	$\delta_{s_3}(t-1)v_{s_3s_1}(t)\phi_{s_1}(t)$	$+$	$\delta_{s_4}(t-1)v_{s_4s_2}(t)\phi_{s_2}(t)$

The max backward scores, ϕ, essentially put a “weight” on each of the models, which tells us how much the portion of the score from that model will influence the overall score. This reflects the fact that, if one model's score for the best value is significantly higher than the other, then any changes to that model's score will have a correspondingly larger effect on the overall score. For example, if the highest-scoring path gets a score of 0.001 from model M_1, and a score of 0.0001 from model M_2, for a total score of 0.0011, then a change to M_1's score, such as increasing it by a factor of 1.1, will have a much larger effect than if the same change were made to model M_2's score.

6.5.3 Subnode Weightings

Thus, the reason that we can't determine which incoming path will maximize the score is that we don't know how much weight to give to the paths through each of the group-node's subnodes. These weights are determined by the max backward scores ϕ. But it's not necessary to know the ϕ values themselves; we only need to know their *relative* values.

For the case where each group node has two subnodes, define R to be the ratio of the two subnodes' ϕ scores:

$$R_g(t) = \phi_{s_1}(t)/\phi_{s_2}(t) \qquad (g = \{s_1, s_2\}) \tag{6.44}$$

Given the value of $R_g(t)$, we can now determine which of the incoming paths will generate the highest score:

$$\begin{aligned}
&\arg\max_{path_i} (\text{score}(path_i)) = \\
&\qquad \arg\max_{path_i} \sum_{s \in g_{t-1,i}} \delta'_s(t-1) v_{ss_1}(t) \phi_{s_1}(t) + \delta'_s(t-1) v_{ss_2}(t) \phi_{s_2}(t) = \\
&\qquad \arg\max_{path_i} \sum_{s \in g_{t-1,i}} \delta'_s(t-1) v_{ss_1}(t) \frac{\phi_{s_1}(t)}{\phi_{s_2}(t)} + \delta'_s(t-1) v_{ss_2}(t) = \\
&\qquad \arg\max_{path_i} \sum_{s \in g_{t-1,i}} \delta'_s(t-1) v_{ss_1}(t) R_g(t) + \delta'_s(t-1) v_{ss_2}(t)
\end{aligned}$$

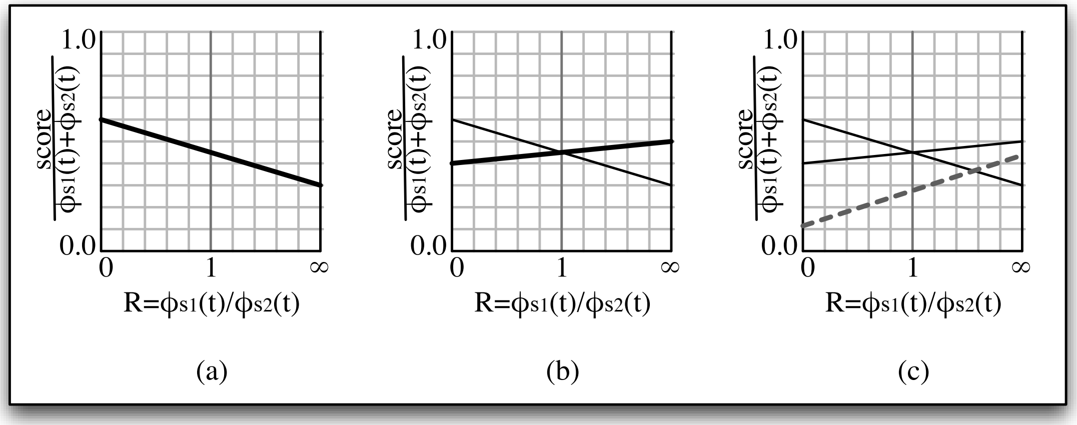


Figure 6.14: **Pruning Candidate Incoming Paths with R.** In graph (a), we plot R vs. the overall score for an incoming path, given that value of R. In graphs (b) and (c), we plot the same function for two more incoming paths. Since the line segment added in graph (c) is not maximal for any value of R, it can be safely pruned.
At the left edge of the graph, where $R = 0$, the score function reduces to $score/\phi_{s_2}(t)$; i.e., when $R = 0$, all of the weight is given to model M_2. At the right edge of the graph, where $R = \inf$, the score function reduces to $score/\phi_{s_1}(t)$, giving all of the weight to model M_1. In the center of the graph, where $R = 1$, equal weight is given to both models.

6.5.4 Pruning Candidate Incoming Paths with R

Of course, there is no no tractable way to calculate $R_g(t)$. But we can make use of the fact that the score of an incoming path depends on this single variable to prune the set of incoming paths under consideration. Figure 6.14 illustrates how this works. Corresponding to each incoming path, we can construct a graph showing the relationship between the value of $R_g(t)$ and the overall score that would be achieved by selecting that path. Figure 6.14 (a) shows such a graph, plotting $R_g(t)$ vs. the overall score of the path (normalized by $\phi_{s_1}(t)+\phi_{s_2}(t)$). Note that this graph is linear, assuming we plot the graph using an x axis where $x = 1 - 1/(R + 1)$.

Thus, corresponding to each incoming path we can plot a single line segment. Figure 6.14 (b) adds the line segment for a second incoming path. Since we are interested in selecting the incoming path that maximizes the overall score, we can now tell that the incoming path added in (a) will be superior to the incoming path added in (b) iff $R < 0.95$ (the crossover point).[5] In Figure 6.14 (c), we add the line segment for a third incoming path. However, this new line segment is not maximal for any value of R, and so it can be safely pruned.

As this example illustrated, we can prune any incoming paths whose corresponding line segment is not maximal for any value of R. As we add more incoming paths, these line segments will form a convex "bowl shaped" top surface, defining the maximal score that can be achieved for different values of R. The more segments become a part of this concave surface, the more likely it becomes that the addition of a new segment will result in the pruning of at least one segment. Thus, in most practical problems, the number of segments in the concave surface should remain relatively small, and the number of paths that we need to keep track of will not grow exponentially.

Extension to More than Two Subnodes

The pruning analysis presented so far applies only to Grouped State Viterbi Graphs where each group-node contains two subnodes. In graphs where group-nodes have more subnodes, we will have more than two ϕ values, so a single R value will not suffice. However, we can make use of the same basic approach, by extending the pruning graph to three or more dimensions. In particular, for a Grouped State Viterbi Graph where each node has at most n subnodes, we will need to construct a graph with $n-1$ independent variables, describing the relative weight given to the different subnodes, and one dependent variable, indicating the resulting score. Each incoming edge will be represented by an $n-1$-dimensional hyperplane on this graph; and the set of hyperplanes that contribute to the maximal score values will form an n-dimensional bowl.

[5] The normalization factor $\phi_{s_1}(t) + \phi_{s_2}(t)$ can be ignored when maximizing the score, since it is a constant value.

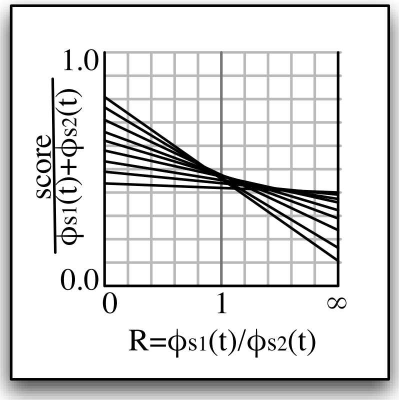


Figure 6.15: **Problematic Case for Pruning**.

6.5.5 Approximate-Best Variant

Although my experiments suggest that the number of line segments or hyperplanes will remain manageable for many real-world problems, there still exist problems for which this pruning approach will not yield any gains. Figure 6.15 illustrates how this can happen. The exact algorithm described in the previous section must keep every incoming path that is maximal for *any* value of R, even if the range of R values for which it is maximal is very small. Figure 6.15 shows how it is possible to construct a large number of line segments, each of which is maximal for only a very small range of R values. In such situations, the exact pruning algorithm is forced to keep track of all incoming paths; and the number of incoming paths can grow to be exponential.

In these cases, it may still be possible to find a value whose score is close to the best value's score by selectively pruning incoming paths. In particular, when we prune an incoming path that is maximal for some value of R, we may be throwing away the best path; but we can still determine an upper bound on how much that pruning will lower the score of the value we find, compared to the optimal value. To understand how, see Figure 6.16, where there are three incoming paths contributing to the pruning graph. Consider the case where we prune path b. In the worst case,

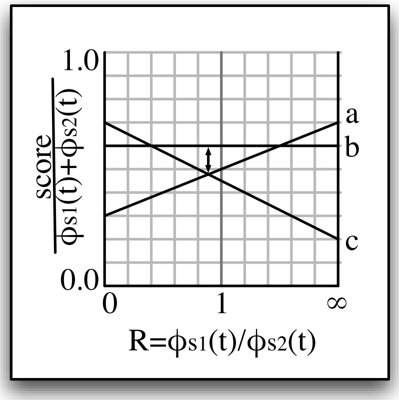


Figure 6.16: **Using Pruning to find the Approximate-Best Value**.

the actual best-scoring path would include path b, and would occur at the R value where b's line segment is the greatest distance from any other line segment. This occurs at around $R = 0.8$, where the line segments for a and c cross. In this worst-case scenario, the score of the best found value will drop by the difference between b's value and a or c's value at $R = 0.8$.

Thus, as long as we restrict ourselves to only prune paths whose corresponding line segments are maximal in a small range of R values, and whose value is not much higher than the surrounding segments, we can limit the potential loss in score incurred by pruning.

6.6 Linear Voting for Models with Differing Problem Decompositions

Section 6.5 showed how multiple models that all use the some encoding can be combined into a single model via linear voting. In this Section, I will show how this approach can be extended to the case of combining multiple models with different encodings. We will use the same basic approach that we used to perform log-linear voting for models with different encodings. In particular, we will convert each model's Viterbi graph to a Canonical-Encoding GSVG; and then combine those GSVGs into a single merged GSVG. However, we will use a different method of combination than we used with log-linear voting; and we will find the highest-scoring output value by searching for the group sequence with the highest score, rather than the simple path with the highest score.

As was the case with log-linear voting, we need to "align" the individual models' Viterbi graphs before we can combine them into a single graph. We do this using exactly the same technique that we used for log-linear voting – namely, we use the FST that represents each model's encoding to translate its Viterbi graph to a corresponding Canonical-Encoding GSVG. See Section 6.4.3 for a detailed description of this algorithm.

Once each model's Viterbi Graph has been converted to a Canonical-Encoding Grouped State Viterbi Graph, we can perform weighted voting by simply merging the individual Canonical-Encoding GSVGs into a single graph. In particular, we construct a new merged GSVG as follows:

1. States in the merged GSVG are tuples pairing a model with a single state from that model's Canonical-Encoding GSVG:

$$S^{\widehat{M}} = \left\{ \langle M_i, s \rangle : s \in S^{M_i} \right\} \tag{6.45}$$

2. The state grouping function for the merged GSVG simply delegates to the grouping functions for the individual model's GSVGs:

$$g^{\widehat{M}}(\langle M_i, s \rangle) = g^{M_i}(s) \tag{6.46}$$

3. The Grouped-State Viterbi Graph transition scores are copied from the individual models' GSVGs, with the initial edge weights modified to account for

the model weights w_i:

$$v^{\widehat{M}}_{\langle M_i,s\rangle}(1) = w_i v^{M_i}_s(1) \tag{6.47}$$

$$v^{\widehat{M}}_{\langle M_i,s\rangle\langle M_i,s'\rangle}(t) = \left(v^{M_i}_{ss'}(t)\right) \tag{6.48}$$

$$v^{\widehat{M}}_{\langle M_i,s\rangle\langle M_j,s'\rangle}(t) = 0 \qquad (\forall i \neq j) \tag{6.49}$$

(Note that this is almost identical to the algorithm we used in Section 6.5 to construct the combined GSVG when performing linear voting on models with identical problem decompositions.)

Once this merged graph has been created, we can apply the algorithms described in Section 6.5.1 to find the highest-scoring path through group nodes. Decoding the corresponding sequence of canonical tags will generate the highest-scoring output value under the linear-voting model.

6.7 Experiments

In order to evaluate the effectiveness with which both linear and log-linear voting techniques can take advantage of the differences between different problem decompositions, I constructed voting models for three sequence prediction tasks: noun phrase chunking, bio-entity recognition, and semantic role labeling. In evaluating these voting models, we are interested in answering three questions:

1. Under what conditions do the voting models improve performance over the individual models?

2. How much performance do they give?

3. For linear voting, does the algorithm described in 6.5.1 allow us to tractably compute the exact best output value?

6.7.1 Noun Phrase Chunking Experiments

The NP chunking experiments all use the same training and testing corpora and feature set that were used for the NP chunking experiments described in Sections 4.2 and 4.8.1. In particular, testing and training were performed using the noun phrase chunking corpus described in Ramshaw & Marcus (1995) (Ramshaw and Marcus, 1995); and the feature set used for all NP chunkers is shown in Figure 4.3.

Three NP chunking experiments were performed, to help evaluate the effect of voting for different system combinations. The first experiment uses voting to combine five NP chunking systems, each of which uses a fairly common (but different) encoding scheme for NP chunks. The second experiment uses voting to combine a first-order NP chunker with a second-order NP chunker. The final NP chunking experiment combines four models using automatically-learned encodings with one model using a hand-crafted encoding.

Voting Between Standard Encodings

The first experiment uses voting to combine noun phrase chunking systems that use five different encodings for chunks, shown in Figure 6.17. Weights for each model were chosen from the values $(0.25, 0.5, 0.75, 1.0)$ (and then normalized to sum to one), and were selected to optimize the voted model's score on the held-out data set.

The complexity of the algorithm stayed fairly low for every sentence in the test corpus. The largest number of paths that needed to be tracked per node (i.e., surfaces in the "concave bowl") was 582, and for the average sentence, the number of paths per node was 47. Table 6.1 compares the performance of the two global voting system to each of the individual systems, as well as a local voting system that took the best output from each system, converted it to IOB1, and performed voting over individual IOB1 tags (similar to (Shen and Sarkar, 2005)). All three voting systems outperform the individual systems, with the log-linear voting system performing best (though the differences between the three voting systems are not

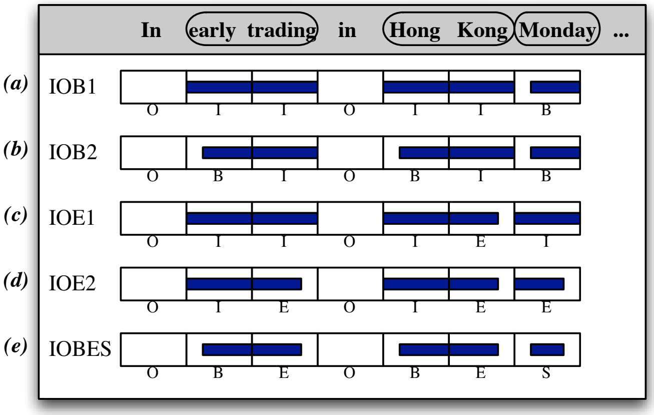


Figure 6.17: **Chunk Encodings**. `O` marks words that are outside a chunk, and `I` marks words that are inside a chunk. IOB1 marks boundaries with `B` on the first word of the second chunk; IOB2 marks the first word of every chunk with `B`; IOE1 marks boundaries with `E` on the last word of the first chunk; and IOE2 marks the last word of every chunk with `E`. IOBES uses `B` and `E` like IOB2 and IOE2; and uses `S` to mark singleton chunks.

statistically significant).

Voting Between First and Second Order Models

The second experiment used voting to combine a first order NP Chunker with a second order NP Chunker. Both systems used the IOB1 encoding. Weights for each model were chosen from the values $(0.25, 0.5, 0.75, 1.0)$ (and then normalized to sum

Model	Precision	Recall	F-Score	
IOB1	93.6	93.8	93.7	
IOB2	93.7	93.9	93.8	
IOE1	93.6	93.8	93.7	
IOE2	93.4	94.0	93.7	
IOBES	93.8	94.0	93.9	
Local Voting	94.1	94.0	94.1	$\star$
Linear Voting	94.2	94.3	94.2	$\star$
Log-linear Voting	94.4	94.3	94.3	$\star$

Table 6.1: **Voting Between Standard NP Chunking Encodings**.
$\star$: score is significantly different from the five individual model scores.

Model	Precision	Recall	F-Score
1^{st} order IOB1	93.6	93.8	93.7
2^{nd} order IOB1	94.4	94.3	94.3
Local Voting	94.4	94.3	94.1
Linear Voting	94.4	94.3	94.3
Log-linear Voting	94.4	94.4	94.3

Table 6.2: **Voting Between First and Second Order NP Chunking Models**.

to one), and were selected to optimize the voted model's score on the held-out data set. Once again, the exact algorithm was tractable for every example in the test corpus. The largest number of paths that needed to be tracked per node was 187, and for the average sentence, the number of paths per node was 12. However, as is shown in Table 6.2, none of the voting systems managed to outperform the second order model. This result most likely reflects the fact that the first order model did not contribute any new information relative to the second order model.

Voting Between Hand-Crafted and Automatically Learned Encodings

The final NP chunking experiment used voting to combine a model based on the hand-crafted encoding described in Section 4.2 with four models based on encodings that were generated using the hill-climbing algorithm described in Chapter 4. Weights for each model were chosen from the values $(0.25, 0.5, 0.75, 1.0)$ (and then normalized to sum to one), and were selected to optimize the voted model's score on the held-out data set.

The four automatically-learned encodings were generated by four successive runs of the hill-climbing system. Each run of the hill-climbing system ran for 200 iterations, after which the encoding that yielded the highest performance on the held-out data was selected. Because the hill-climbing system randomly chooses which FST-modifying operations to apply, and because the search space of possible FSTs is very large, each of these four automatically-learned encodings was different. (However, they did share a number of characteristics, such as the use of transformations that

Model	Precision	Recall	F-Score	
Hand-Crafted Encoding	94.2	94.2	94.2	
Learned Encoding (Run 1)	94.4	94.6	94.5	
Learned Encoding (Run 2)	94.5	94.7	94.6	
Learned Encoding (Run 3)	94.6	94.6	94.6	
Learned Encoding (Run 4)	94.8	94.3	94.5	
Linear Voting	94.9	94.8	94.9	⋆
Log-linear Voting	95.1	94.8	95.0	⋆

Table 6.3: **Voting Between Hand-Crafted and Automatically Learned NP Chunking Encodings**.

⋆: score is significantly different from the five individual model scores.

extend the amount of history information encoded in tags.)

The exact algorithm was tractable for every example in the test corpus. The largest number of paths that needed to be tracked per node was 1018, and for the average sentence, the number of paths per node was 56. Table 6.3 compares the performance of the two global voting systems to each of the individual systems. We can see that the performance improvements that come from using voting are not redundant with the performance improvement that comes from using customized encodings. Once again, the log-linear voting system achieves a slightly higher score than the linear voting system, although the difference is not statistically significant.

6.7.2 Bio-Entity Recognition Experiment

The bio-entity recognition experiment used the same training and testing corpora and feature set that were used for the bio-entity recognition experiments described in Section 4.8.1. In particular, testing and training were performed using the JNLPBA bio-entity recognition corpus; and the feature set used for all recognizers is shown in Figure 4.13. Weights for each model were chosen from the values $(0.25, 0.5, 0.75, 1.0)$ (and then normalized to sum to one), and were selected to optimize the voted model's score on the held-out data set.

This experiment used voting to combine five models based on encodings that

Model	Precision	Recall	F-Score	
Learned Encoding (Run 1)	73.2	66.8	69.9	
Learned Encoding (Run 2)	72.8	67.0	69.8	
Learned Encoding (Run 3)	73.4	66.8	70.0	
Learned Encoding (Run 4)	73.6	66.9	70.1	
Learned Encoding (Run 5)	71.9	66.4	69.1	
Linear Voting	73.6	67.5	70.4	$\star$
Log-linear Voting	73.5	67.6	70.4	$\star$

Table 6.4: **Voting Between Automatically Learned Bio-Entity Recognition Encodings**.

$\star$: score is significantly different from the five individual model scores.

were generated using the hill-climbing algorithm described in Chapter 4. These five encodings were generated by four successive runs of the hill-climbing system. Each run of the hill-climbing system ran for 150 iterations, after which the encoding that yielded the highest performance on the held-out data was selected.

The exact algorithm was tractable for every example in the test corpus. The largest number of paths that needed to be tracked per node was 1426, and for the average sentence, the number of paths per node was 32. Table 6.4 compares the performance of the two global voting system to each of the individual systems. Both voting systems yield similar results, and outperform each of the individual systems.

6.7.3 Semantic Role Labeling Experiment

The final experiment evaluates the effectiveness of the voting methods for the semantic role labeling system described in Chapter 5. As was discussed in Chapter 5, long-distance constraints are more important for SRL than they are for the other two problems considered so far.

For this experiment, I used the same training and testing corpora and feature set that were used for the experiments described in Chapter 5. In particular, testing and training were performed using the PropBank corpus, with section 23 used for testing and section 24 used as a held-out corpus. The features used for the SRL

Model	Precision	Recall	F-Score	
Learned Encoding (Run 1)	78.6	73.7	76.1	
Learned Encoding (Run 2)	78.7	73.3	75.9	
Learned Encoding (Run 3)	78.6	74.0	76.2	
Learned Encoding (Run 4)	78.6	73.1	75.8	
Learned Encoding (Run 5)	78.3	74.0	76.1	
Linear Voting	78.7	75.4	77.0	$\star$
Log-linear Voting	78.9	75.7	77.2	$\star\dagger$

Table 6.5: **Voting Between Automatically Learned SRL Encodings**.
$\star$: score is significantly different from the five individual model scores.
$\dagger$: score is significantly different from the linear voting score.

system are described in Figure 5.3. Weights for each model were chosen from the values $(0.25, 0.5, 0.75, 1.0)$ (and then normalized to sum to one), and were selected to optimize the voted model's score on the held-out data set.

This experiment used voting to combine five models based on encodings that were generated using the hill-climbing algorithm described in Chapter 4. These five encodings were generated by four successive runs of the hill-climbing system. Each run of the hill-climbing system ran for 150 iterations, after which the encoding that yielded the highest performance on the held-out data was selected.

The exact algorithm was tractable for every example in the test corpus. The largest number of paths that needed to be tracked per node was 472, and for the average sentence, the number of paths per node was 36. Table 6.5 compares the performance of the two global voting system to each of the individual systems. Both voting systems outperform the individual systems by a wide margin, yielding an increase in f-score of 1.0% and 0.8% relative to the highest-scoring individual model. The fact that this performance improvement is larger than the improvements we saw in the other experiments may reflect the fact that long-distance dependencies and constraints are more important to the SRL task; and that the different learned encodings may be capturing different long-distance constraints.

6.8 Discussion

We have seen that it is possible to use both linear and log-linear voting to combine structured prediction models that use different problem decompositions. In order to tractably combine the scores from the individual models, it was important to first find a way to align the graphs that the individual models used to encode their probability distributions. In the case of sequence prediction problems, we saw that this alignment can be performed by making use of the finite state transducers that relate the encodings to one another. I believe that this basic technique could be extended to other structured prediction problems, such as parsing; but that to do so, we would need to design new alignment algorithms specific to those problem domains. This task can be made significantly easier if we restrict the ways in which models can be related to one another (e.g., if we only allow models that subdivide the canonical model's tags).

We have also seen that the combined models produced by linear and log-linear voting can yield performance improvements over the individual models, especially if there is variety in the type of information encoded by the different models.

Our results suggest that log-linear voting may yield more of a performance gain than linear voting. One possible explanation for this result is based on the fact that the probabilities assigned to entire output structures by individual models can vary widely. For example, the probabilities assigned to the best path by a simple IOB1 NP chunker can range from 0.5 all the way down to 1e-12. Consider a case where we are combining two individual systems; and the probabilities that they assign to their 5 most likely outputs are:

Model 1

Output Value	Probability
IOIIBIIOI	0.0062
IOIIIIIOI	0.0040
IOIOIIIOI	0.0036
OOIIBIIOI	0.0034
OOIIBIIOO	0.0018

Model 2

Output Value	Probability
IOIIIIIOI	0.00034
IOIOIIIOI	0.00016
IOIIBIIOI	0.00014
OOIIBIIOI	0.00008
IOIIBBIOI	0.00008

Because the output scores generated by the first model are over an order of magnitude higher than the scores generated by the second model, the combined scores generated by the linear voting method (which uses an arithmetic mean to combine scores) will primarily just reflect the scores given by the first model. Thus, the highest scoring value using linear voting is IOIIBIIOI (with a voted score of 0.0032). In contrast, because the log-linear method generates combined scores by multiplying together the individual models' scores, the combined scores will reflect information about the relative differences between individual outputs' scores from both models' probability distributions. Thus, the highest scoring value using log-linear voting is IOIIIIIOI (with a weighted score of 0.0012).

This suggests that log-linear voting may be a more effective technique for combining individual models' scores when we are interested in preserving information contained in the relative rankings given by those individual models, as opposed to their overall absolute probability estimates.

Chapter 7

Conclusions

This dissertation has focused on supervised machine learning systems that perform structured prediction by decomposing the overall prediction problem into a set of simpler sub-problems, with well-defined and well-constrained interdependencies between those sub-problems. For such systems, we have seen that the manner in which the prediction problem is decomposed into sub-problems – and the way in which those sub-problems are divided into equivalence classes – can affect the ability of the machine learning system to accurately model a given problem domain.

We have shown that *reversible output encoding transformations* can provide a powerful and effective tool for exploring different problem decompositions. Each output encoding transformation corresponds to a specific way of subdividing the overall prediction problem. By defining a large class of possible encoding transformations, we can explore a wide variety of problem decompositions. And since the problem decompositions are represented using the transformed output encoding, we can explore a wide variety of possible model structures without needing to modify the underlying machine learning system.

7.1 The Effects of Transforming Encodings

Transforming the encoding that is used to represent the output values of a task, and by extension transforming the decomposition of the task into sub-tasks, affects the accuracy with which machine learning methods can model the task and predict the correct output values for new inputs. In particular, we can divide the effects that output encoding transforms have on the machine learning system's ability to model a task into two general classes: local effects and global effects.

Local effects reflect changes in the model's ability to accurately model individual sub-problems. These effects depend primarily on the coherence of the classes that are defined by individual output tags, on how easily different output tags can be discriminated from one another, and on the amount of training data that is available for each local problem.

Global effects reflect changes in the model's ability to accurately model dependencies between different sub-problems. These effects depend primarily on the information content of the output tags. We can divide the transformations that have global effects into two groups: fixed-window and long-distance transformations. Fixed-window transformations allow the model to capture dependencies between output tags that are not immediately adjacent to a given sub-problem, but are still within a fixed window of that sub-problem. The transformations that generate higher order Markov Models, discussed in Section 2.4, are an example of fixed-window transformations. On the other hand, long-distance transformations allow information about output tags to propagate over arbitrary distances within the output structure. The transformations described in Section 5.6 are an example of long-distance transformations.

Although it can be useful to divide the effects that a transformation has on a machine learning system into local and global effects, it should be noted that there are almost always interactions between these effects; and any given transformation will almost always have both local and global effects. It can, however, still be useful to think about output encoding transforms in these terms, in order to help decide which transforms may be useful, or why a transform might have the effects that it does. In particular, in cases where the individual sub-problems of a structured prediction task seem to lack coherence (e.g., PropBank argument labels), one should consider making use of transforms whose local effects can increase the labels' coherence; and in cases where non-local dependencies between individual pieces of a structured prediction problem is important (e.g., SRL), one should focus on transforms with global effects that could allow those dependencies to be learned.

7.2 Effect of the Machine Learning System

Given any machine learning system, the application of output encoding transformations will affect that system's ability to model a problem domain. However, the effects that an output encoding transform has can depend on the properties of the machine learning system. In particular, some machine learning systems have properties that may make it more likely that they will be able to take advantage of output encoding transformations to model a problem domain more accurately. This dependence on the properties of the machine learning system were especially demonstrated in Section 4.9, where we combined the hill-climbing system that searches for an improved output encoding with an underlying HMM model. In this section, we describe two properties of a machine learning system that may improve its ability to take advantage of output encoding transformations.

First, the machine learning system should have some way to make generalizations over multiple labels. This ability allows us to create a richer space of labels without splitting our training data excessively (which can result in decreased performance due to sparse data problems). In the Maximum Entropy and CRF models, we can allow the machine learning system to generalize over labels by taking advantage of the fact that features are defined as functions of both the input and the output, and in particular by using subset features. In other models, it may be possible to use some form of "backoff" to allow for generalizations over multiple labels. However, the experiments in Section 4.9 did not show any improvement when backoff was added to the HMM variant of the hill-climbing system.

Second, machine learning systems that are trained using global optimization techniques may be better able to take advantage of transformed output encodings. Models that are not trained globally tend to make strong assumptions about the problem domain, and specifically about the independence of different decisions about subproblems. In this context, output encoding transformations can often end up adding new assumptions, which may not be accurate for the problem domain; and these new inaccurate assumptions may decrease overall system performance. In contrast, machine learning systems that are trained using global optimization techniques do not rely as strongly on such assumptions; instead, they simply find the set of parameter values that allow them to best model the training data. Thus, systems that use global optimization techniques may be better able to adapt to and overcome any negative effects caused by changing the model structure.

7.3 Relationship to Previous Node-Splitting Work

The output encoding transformations that I have presented build on previous work that focused on adding contextual information by splitting tags, labels, or nodes (Johnson, 1998; Collins, 1999; Klein and Manning, 2003a; Matsuzaki et al., 2005; Petrov et al., 2006)]. The transformational framework that I have presented significantly extend the type of output encoding transformations that can be considered, beyond simple node splitting, to include transformations such as reordering, recombination, and restructuring. These more advanced transformations can help to improve the coherence of the individual labels, and to properly capture non-local dependencies.

7.4 Summary of Original Contributions

In this dissertation, I have demonstrated that transforming the output structure that is used by a machine learning system to encode structured output values can improve its ability to model the problem domain. In particular, I examined four tasks: part of speech tagging, NP chunking, bio-entity recognition, and semantic role labeling; and showed that output encoding transformations could be used to improve performance for each of these tasks.

I have also presented a novel hill-climbing algorithm that can be used to automatically search for problem decompositions that improve performance. This system uses finite state transducers to provide a concrete representation for output encoding transformations, and uses a carefully defined set of modification operations to explore a variety of different problem decompositions.

In Chapter 5, I have presented a novel architecture for performing semantic role labeling, which uses a single structured prediction model to jointly predict all of a given verb's arguments. I also showed how this architecture could be adapted, using output encoding transformations, to make use of dependency information between different verb arguments. After using the hill-climbing algorithm to automatically search for an improved output encoding, I generalized the results to produce a hand-crafted output encoding transformation which yielded a further improvement in the system's performance.

I have demonstrated that output encoding transformations can affect the performance of a machine learning system in two ways: by making local sub-problems more coherent; and by modifying the set of dependencies between different sub-problems that the model can learn. The first ("local") type of effect is demonstrated most clearly by experiments described in Chapter 3, which made use of the SemLink mapping to improve the coherence of PropBank labels. The second ("global") type of effect is demonstrated most clearly by the experiments described in Chapter 5, which showed that dependencies between arguments in an SRL system can be learned by augmenting the tag set used to model the arguments.

Finally, I have presented techniques for combining sequence prediction models that use different problem decompositions using both linear and log-linear voting. These techniques both make use of finite state transducers that map between the individual models' encodings to align their Viterbi graphs. The linear voting technique also makes use of a modified version of the Viterbi decoding algorithm that uses careful pruning to make it possible to perform exact prediction in the common case, and approximate prediction in the general case. I have presented experimental evidence that suggests that, although both voting methods can take advantage of differences between individual models to improve performance, log-linear voting may yield a larger performance gain for structured prediction tasks.

Appendix A

Finding the Optimal Group Sequence in a Grouped State Viterbi Graph is NP-Hard

As mentioned in Section 6.5.1, the problem of finding the optimal group sequence in a Grouped-State Viterbi Graph is NP-hard in the general case. This appendix presents a proof of that result. It is loosely based on the proof in (Casacuberta and Higuera, 2000), which considers the analogous problem of finding the most probable string output for a stochastic random grammar.

To show that finding the optimal group sequence is NP-hard, we show how the NP-complete problem of 3-SAT can be reduced to this problem in polynomial time. 3-SAT is the problem of determining whether there is any assignment to a fixed set of variables $\{v_1, \ldots, v_n\}$ that makes a given boolean equation true. The boolean equation is restricted to have the form:

$$(x_{1,1} \vee x_{1,2} \vee x_{1,3}) \wedge \ldots \wedge (x_{k,1} \vee x_{k,2} \vee x_{k,3}) \tag{A.1}$$

where $x_{i,j} \in (\{v_1, \ldots, v_n\} \cup \{\overline{v_1}, \ldots, \overline{v_n}\})$.

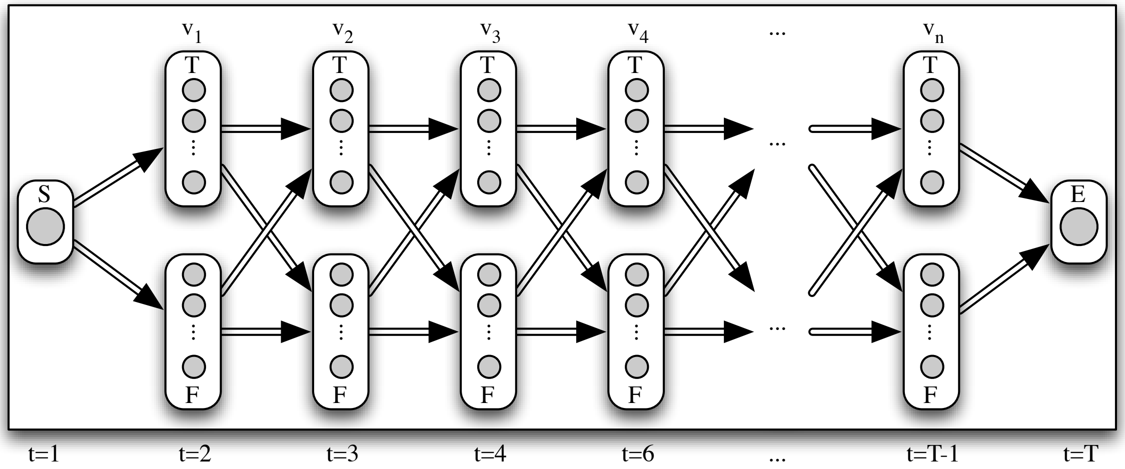


Figure A.1: **Basic Architecture of Graph Corresponding to 3-SAT**. Each variable v_i is represented by a single time slice $t = i + 1$. Two groups, "T," and "F," are used to represent the variables' truth values. Arrows represent paths through group nodes (as opposed to paths through individual subnodes).

In order to transform 3-SAT into the problem of finding an optimal group sequence through a Grouped-State Viterbi Graph, we will show how a graph can be constructed from the boolean equation, whose highest scoring group path will have a score of k if the boolean equation is satisfiable; and of less than k if it is not. The basic structure of this Grouped-State Viterbi Graph is shown in Figure A.1.

Each variable v_i is represented by a single time slice $t = i + 1$. Within each time slice, the graph contains two groups "T," and "F," corresponding to the boolean values true and false. Thus, each group path through the graph corresponds directly to an assignment of values to variables.

We will take advantage of this fact by creating a separate subgraph for each clause in the boolean expression, that will include a single complete path with score 1 through any sequence of groups that satisfies the clause; but will not contain any complete paths through group sequences that do not satisfy the clause. Since the score of a group sequence is equal to the sum of the scores of all paths through the group sequence, the total score will be equal to the number of clauses made true by the group sequence. Thus, the total score of a group sequence will only be equal to

k if all k clauses are made true by the group sequence; and the maximum score of all group sequences will only be k if there exists such a group sequence (corresponding to a variable assignment that satisfies the original 3-SAT problem).

Figure A.2 illustrates how the subgraph corresponding to an individual clause is constructed, using the clause $(v_1 \vee \overline{v_3} \vee v_5)$ as a concrete example. First, we construct a subgraph of the form shown in Figure A.2*(a)*, which contains two sets of subnodes:

- The unshaded nodes, s, a_t, and b_t, which are fully connected (i.e., every node at each time slice t is connected to every node at time slice $t{+}1$); and which include the start node. These nodes will be used when we have not yet determined if the clause is satisfied.
- The shaded nodes, x_t, y_t, and e, which are fully connected; and which include the end node. These nodes will be used once we have determined that the clause is satisfied.

Then in Figure A.2*(b)*, we replace two of the edges with new edges from the unshaded nodes to the shaded nodes for each variable literal in the clause, at the locations in the graph where we might determine that the clause is satisfied:

- If v_i appears as a positive literal in the clause, then replace edges $\langle a_i \rightarrow a_{i+1} \rangle$ and $\langle a_i \rightarrow b_{i+1} \rangle$ with edges $\langle a_i \rightarrow x_{i+1} \rangle$ and $\langle a_i \rightarrow y_{i+1} \rangle$.[1]
- If v_i appears as a negative literal in the clause, then replace edges $\langle b_i \rightarrow a_{i+1} \rangle$ and $\langle b_i \rightarrow b_{i+1} \rangle$ with edges $\langle b_i \rightarrow x_{i+1} \rangle$ and $\langle b_i \rightarrow y_{i+1} \rangle$.[2]

As a result, the graph will contain a path from the start node to the end node for exactly those group sequences that correspond to variable assignments that make the clause true.

Finally, we combine the subgraphs for each clause into a single Grouped State Viterbi Graph, merging the groups from the individual subgraphs; and find the

[1] If $i = T - 1$, then add a single edge $\langle a_i \rightarrow e \rangle$ instead.

[2] If $i = T - 1$, then add a single edge $\langle b_i \rightarrow e \rangle$ instead.

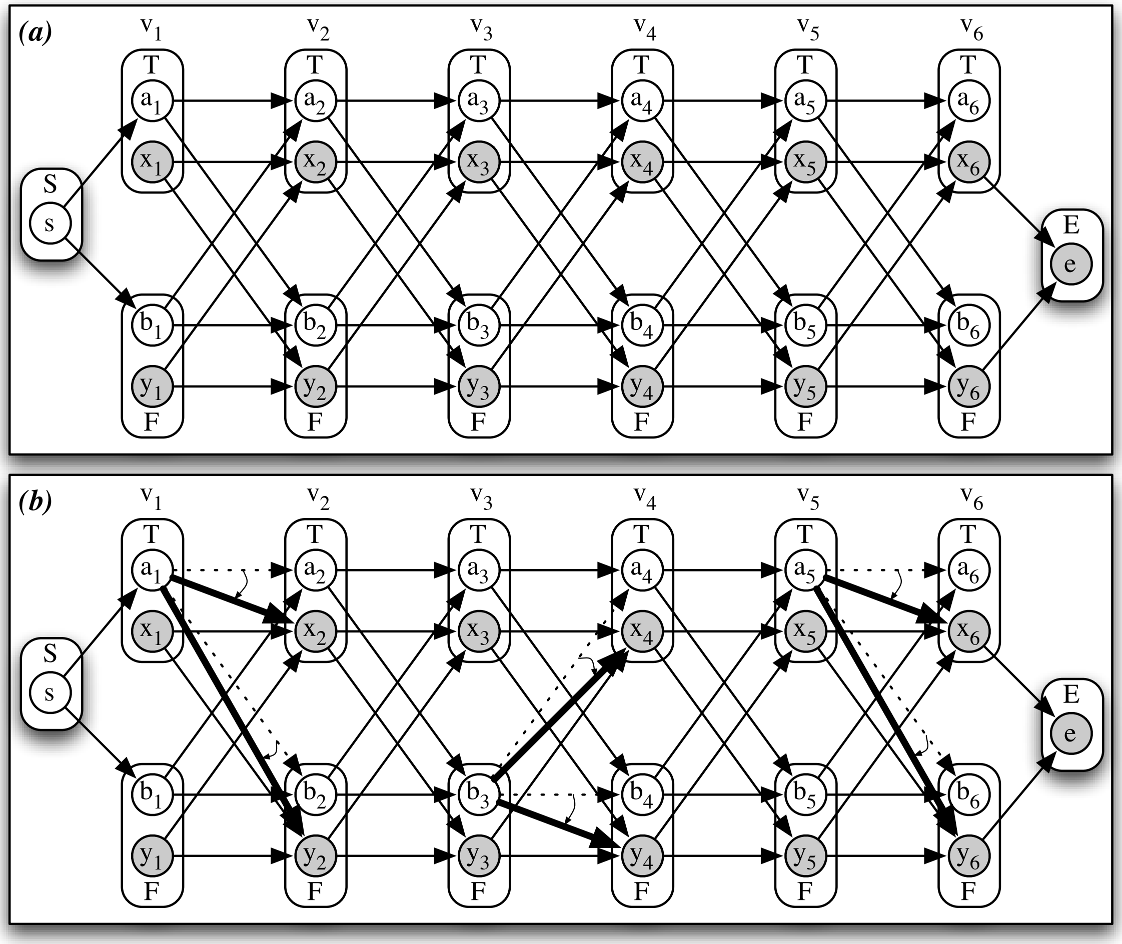


Figure A.2: **Construction of the Subgraph Corresponding to a Single Clause**. This figure shows how the subnode corresponding to the clause $(v_1 \vee \overline{v_3} \vee v_5)$ is constructed (with n=6 variables). In step *(a)*, we create two disconnected subgraphs. The first subgraph (unshaded nodes) contains a single subnode in every group node except "E", and is fully connected. The second subgraph (shaded nodes) contains a single subnode in every group node except "S", and is fully connected. In step *(b)*, we add edges from the first graph to the second graph, corresponding to the variable assignment expressions that will make the clause true (added edges are shown in bold; removed edges are shown as dotted arrows). As a result, the graph will contain a path from the start node to the end node for exactly those group sequences that correspond to variable assignments that make the clause true.

highest scoring group path through the combined graph. If the score of this group path is k, then the corresponding 3-SAT problem is satisfiable; and if the score of this group path is less than k, then the corresponding 3-SAT problem is not satisfiable.

We have shown that the 3-SAT problem can be solved by constructing a corresponding Grouped-State Viterbi Graph (in polynomial time), and evaluating the score of the highest scoring path. Thus, since the 3-SAT problem is NP-hard, the problem of finding the highest scoring path in a Grouped-State Viterbi Graph must also be NP-hard.

Bibliography

Ethem Alpaydin, 2004. *Introduction to Machine Learning*, chapter 7. The MIT Press.

Daniel M. Bikel. 2004a. A distributional analysis of a lexicalized statistical parsing model. In *Conference on Empirical Methods in Natural Language Processing (EMNLP-2004)*, pages 182–189. `http://www.cis.upenn.edu/~dbikel/software.html`.

Daniel M. Bikel. 2004b. Intricacies of collin's parsing model. *Computational Linguistics*, 30(4).

Xavier Carreras and Lluís Márquez. 2004. Introduction to the conll-2004 shared task: Semantic role labeling. In *Proceedings of CoNLL*.

Xavier Carreras and Lluís Márquez. 2005. Introduction to the conll-2005 shared task: Semantic role labeling. In *Proceedings of CoNLL*.

F. Casacuberta and Colin De La Higuera. 2000. Computational complexity of problems on probabilistic grammars and transducers. In *Grammatical Inference: Algorithms and Applications, 5th International Colloquium, ICGI 2000, Lisbon, Portugal, September 11 - 13, 2000 ; Proceedings*, volume 1891, pages 15–24. Springer, Berlin.

John Chen and Owen Rambow. 2003. Use of deep linguistic features for the recognition and labeling of semantic arguments. In *Conference on Empirical Methods in Natural Language Processing (EMNLP-2003)*, Sapporo, Japan.

Trevor Cohn and Philip Blunsom. 2005. Semantic role labeling with tree conditional random fields. In *Proceeding of the 9th Conference on Computational Natural Language Learning (CoNLL)*.

Trevor Cohn, Andrew Smith, and Miles Osborne. 2005. Scaling conditional random fields using error-correcting codes. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics (ACL-05)*, pages 10–17, Morristown, NJ, USA. Association for Computational Linguistics.

Michael Collins. 1999. *Head-Driven Statistical Models for Natural Language Processing*. Ph.D. thesis, University of Pennsylvania.

Walter Daelemans, Jakub Zavrel, Ko van der Sloot, and Antal van den Bosch. 1999. TiMBL: Tilburg memory based learner, version 2.0, reference guide. Technical Report 99-01, ILK.

S. Dasgupta. 1999. Learning mixtures of gaussians. In *Fortieth Annual IEEE Symposium on Foundations of Computer Science (FOCS)*.

Hal Daumé III. 2004. Notes on CG and LM-BFGS optimization of logistic regression. Paper available at `http://pub.hal3.name#daume04cg-bfgs`, implementation available at `http://hal3.name/megam/`, August.

T. G. Dietterich and G. Bakiri. 1995. Solving multiclass learning problems via error-correcting output codes. *Journal of Artificial Intelligence Research*, 2:263–286.

Christy Doran, Dania Egedi, Beth Ann Hockey, B. Srinivas, and Martin Zaidel. 1994. XTAG system – a wide coverage grammar for english. In *Proceedings of the 15th. International Conference on Computational Linguistics (COLING 94)*, volume II, pages 922–928, Kyoto, Japan.

D. Dowty. 1989. On the semantic content of the notion thematic role. In G. Chierchia, B. Partee, and R. Turner, editors, *Properties, Types, and Meaning, vol. 2*. Kluwer Academic Publishers, Dordrecth.

D. Dowty. 1991. Thematic proto-roles and argument selection. *Language*, 67(3):547–619.

Kai-Bo Duan and S. Sathiya Keerthi. 2005. Which is the best multiclass svm method? An empirical study. In *Multiple Classifier Systems*, pages 278–285. Springer Berlin / Heidelberg.

C. J. Fillmore. 1968. The case for case. In E. Bach and R. Harms, editors, *Universals in Linguistic Theory*. Holt, Rinchart, and Winston, New York.

Daniel Gildea and Julia Hockenmaier. 2003. Identifying semantic roles using Combinatory Categorial Grammar. In *Conference on Empirical Methods in Natural Language Processing (EMNLP-2003)*, pages 57–64, Sapporo, Japan.

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

J. Gruber. 1965. *Studies in Lexical Relations*. Ph.D. thesis, MIT.

Kadri Hacioglu, Sameer Pradhan, Wayne Ward, James H. Martin, and Daniel Jurafsky. 2003. Shallow semantic parsing using support vector machines. Technical report, The Center for Spoken Language Research at the University of Colorado (CSLR).

Chih-Wei Hsu and Chih-Jen Lin. 2002. A comparison of methods for multiclass support vector machines. *IEEE Transactions on Neural Networks*, 13(2):415–425, March.

R. Jackendoff. 1972. *Semantic Interpretation in Generative Grammar.* MIT Press, Cambridge, Massachusetts.

Mark Johnson. 1998. Pcfg models of linguistic tree representations. *Computational Linguistics*, 24(4).

Jin-Dong Kim, Tomoko Ohta, Yoshimasa Tsuruoka, Yuka Tateisi, and Niel Collier. 2004. Introduction to the bio-entity recognition task at jnlpba. In *Proceedings of the COLING 2004 International Joint Workshop on Natural Langauge Processing in Biomedicine and its Applications (NLPBA)*, Geneva, Switzerland.

Dan Klein and Christopher D. Manning. 2003a. Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics.*

Dan Klein and Christopher D. Manning. 2003b. Factored a* search for models over sequences and trees. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence.*

Dan Klein and Christopher D. Manning. 2003c. Fast exact inference with a factored model for natural language parsing. In Suzanna Becker, Sebastian Thrun, and Klaus Obermayer, editors, *Advances in Neural Information Processing Systems 15*, Cambridge, MA. MIT Press.

Taku Kudo and Yuji Matsumoto. 2001. Chunking with support vector machines. In *Proceedings of the North American chapter of the Association for Computational Linguistics (NAACL-01).*

Beth Levin. 1993. *English Verb Classes and Alternations: A Preliminary Investigation.* The University of Chicago Press.

Edward Loper, Szu ting Yi, and Martha Palmer. 2007. Combining lexical resources: Mapping between propbank and verbnet. In *Proceedings of the 7th International Workshop on Computational Linguistics*, Tilburg, the Netherlands.

M. Marcus, G. Kim, M. Marcinkiewicz, R. MacIntyre, A. Bies, M. Ferguson, K. Katz, and B. Schasberger. 1994. The Penn treebank: Annotating predicate argument structure. In *ARPA Human Language Technology Workshop.*

Takuya Matsuzaki, Yusuke Miyoa, and Jun'ichi Tsujii. 2005. Probabilistic cfg with latent annotations. In *Proceedings of the 43rd Annual Conference of the Association for Computational Linguistics (ACL-05).*

Andrew McCallum, Dayne Freitag, and Fernando Pereira. 2000. Maximum entropy Markov models for information extraction and segmentation. In *Proc. 17th International Conf. on Machine Learning*, pages 591–598. Morgan Kaufmann, San Francisco, CA.

Andrew Kachites McCallum. 2002. Mallet: A machine learning for language toolkit. http://mallet.cs.umass.edu.

Mehryar Mohri. 1997. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–311.

Thomas Morton. 2004. *Using Reference Relations to Improve Information Retrieval.* Ph.D. thesis, University of Pennsylvania.

Alessandro Moschitti. 2004. A study on convolution kernel for shallow semantic parsing. In *Proceedings of the 42nd Conference on Association for Computational Linguistic (ACL-04)*, Barcelona, Spain.

José Oncina, Pedro García, and Enrique Vidal. 1993. Learning subsequential transducers for pattern recognition tasks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15:448–458, May.

Martha Palmer, Daniel Gildea, and Paul Kingsbury. 2005. The proposition bank: A corpus annotated with semantic roles. *Computational Linguistics*, 31(1):71–106.

Judea Pearl. 1988. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann.

Slav Petrov, Leon Barrett, Romain Thibaux, and Dan Klein. 2006. Learning accurate, compact, and interpretable tree annotation. In *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the ACL*, pages 433–440.

S. Pradhan, W. Ward, K. Hacioglu, J. Martin, and D. Jurafsky. 2004. Shallow semantic parsing using support vector machines. In *Proceedings of the Human Language Technology Conference/North American chapter of the Association for Computational Linguistics annual meeting (HLT/NAACL-04)*, Boston, MA.

Sameer Pradhan, Kadri Hacioglu, Wayne Ward, H. Martin, James, and Daniel Jurafsky. 2005a. Semantic role chunking combining complementary syntactic views. In *Proceedings of CoNLL-2005.*

Sameer Pradhan, Wayne Ward, Kadri Hacioglu, James Martin, and Dan Jurafsky. 2005b. Semantic role labeling using different syntactic views. In *Proceedings of the Association for Computational Linguistics 43rd annual meeting (ACL-05)*, Ann Arbor, MI.

V. Punyakanok, D. Roth, and W. Yih. 2005. The necessity of syntactic parsing for semantic role labeling. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI-05).*

Lance Ramshaw and Mitch Marcus. 1995. Text chunking using transformation-based learning. In David Yarowsky and Kenneth Church, editors, *Proceedings of the Third Workshop on Very Large Corpora*, pages 82–94, Somerset, New Jersey. Association for Computational Linguistics.

S. Russell and P. Norvig. 1995. *Artificial Intelligence: A Modern Approach.* Prentice Hall.

Karin Kipper Schuler. 2005. *VerbNet: A broad-coverage, comprehensive verb lexicon.* Ph.D. thesis, University of Pennsylvania.

Fei Sha and Fernando Pereira. 2003. Shallow parsing with conditional random fields. In *Proceedings of the Human Language Technology Conference/North American chapter of the Association for Computational Linguistics annual meeting (HLT/NAACL-03)*, pages 134–141.

Hong Shen and Anoop Sarkar. 2005. Voting between multiple data representations for text chunking. In *Advances in Artificial Intelligence: 18th Conference of the Canadian Society for Computational Studies of Intelligence*, May.

Andrew Smith and Miles Osborne. 2007. Diversity in logarithmic opinion pools. *Lingvisticae Investigationes*, 30(1):27 – 47, July.

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 of Computational Linguists (ACL-05).*

P. Smyth. 1998. Belief networks, hidden Markov models, and Markov random fields: a unifying view. *Pattern Recognition Letters.*

Andreas Stolcke and Stephen Omohundro. 1993. Hidden Markov Model induction by Bayesian model merging. In C. L. Giles, S. J. Hanson, and J. D. Cowan, editors, *Advances in Neural Information Processing Systems 5.* Morgan Kaufman, San Mateo, Ca.

Charles Sutton and Andrew McCallum. 2005. Joint parsing and semantic role labeling. In *Proceeding of the 9th Conference on Computational Natural Language Learning (CoNLL)*, pages 225–228.

Charles Sutton and Andrew McCallum. 2006. An introduction to conditional random fields for relational learning. In Lise Getoor and Ben Taskar, editors, *Introduction to Statistical Relational Learning*. MIT Press. To appear.

Cynthia A. Thompson, Roger Levy, and Christopher D. Manning. 2003. A generative model for semantic role labeling. In *Proceedings of ECML-2003*.

Erik Tjong Kim Sang and Jorn Veenstra. 1999. Representing text chunks. In *Proceedings of the European chapter of the Association for Computational Linguistics annual meeting (EACL-1999)*, Bergen. Association for Computational Linguistics.

Erik Tjong Kim Sang. 2000. Noun phrase recognition by system combination. In *Proceedings of BNAIC*, Tilburg, The Netherlands.

Kristina Toutanova, Aria Haghighi, and Christopher D. Manning. 2005. Joint learning improves semantic role labeling. In *Proceedings of the Association for Computational Linguistics 43rd annual meeting (ACL-05)*, Ann Arbor, MI.

Kristina Toutanova, Aria Haghighi, and Christopher Manning. 2008. A global joint model for semantic role labeling. *Computational Linguistics*, 34(2).

Nianwen Xue and Martha Palmer. 2004. Calibrating features for semantic role labeling. In Dekang Lin and Dekai Wu, editors, *Conference on Empirical Methods in Natural Language Processing (EMNLP-2004)*, pages 88–94, Barcelona, Spain, July. Association for Computational Linguistics.

Szu-ting Yi and Martha Palmer. 2004. Pushing the boundaries of semantic role labeling with svm. In *Proceedings of the International Conference on Natural Language Processing*.

Szu-ting Yi and Martha Palmer. 2005. The integration of syntactic parsing and semantic role labeling. In *Proceedings of CoNLL-2005*.

Szu-ting Yi, Edward Loper, and Martha Palmer. 2007. Can semantic roles generalize across genres? In *Proceedings of the Human Language Technology Conference/North American chapter of the Association for Computational Linguistics annual meeting (HLT/NAACL-07)*.