ENCODING STRUCTURED OUTPUT VALUES

Edward Loper

A DISSERTATION PROPOSAL

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

2007

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, affection 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.

For my dissertation, I plan to apply automatic and semi-automatic methods to the problem of finding optimal problem decompositions, in the context of three specific systems (one chunking system and two semantic role labeling systems). Additionally, I plan to evaluate a novel approach to voting between multiple models when each model uses a different problem decomposition, which I describe in Chapter 7.

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,

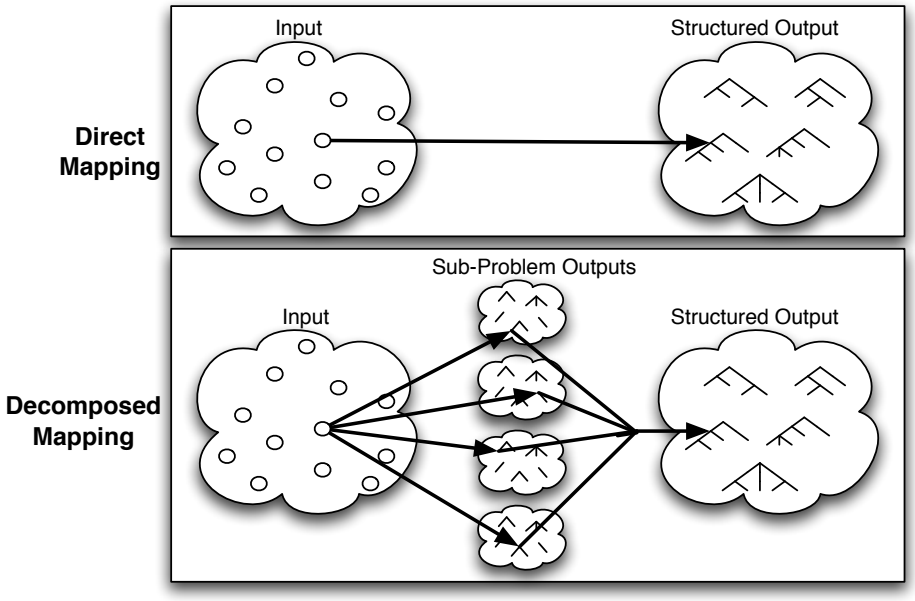


Figure 1.1: **Decomposing a Structured-Output Mapping**. In problems with structured outputs, the large number of possible output values usually makes it impractical to learn the direct mapping from inputs to outputs (top). Instead, the problem can be decomposed into a set of simpler sub-problems; and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

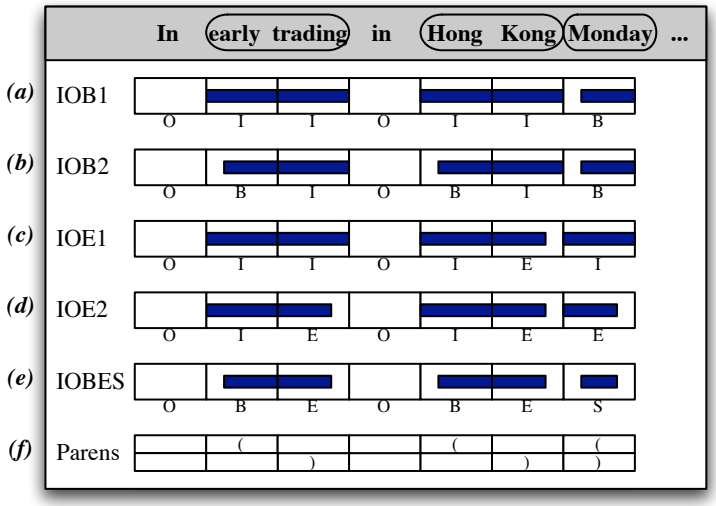


Figure 1.2: **Example Output Encodings: NP Chunking**. This figure shows six different encodings for the task of finding all "noun phrase chunks" (circled) in a sentence. Encodings *(a)-(e)* assign a tag to each word, indicating how that word is chunked; the meaning of each tag is indicated by the shaded 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 2.2.1 for a more detailed explanation of these encodings.

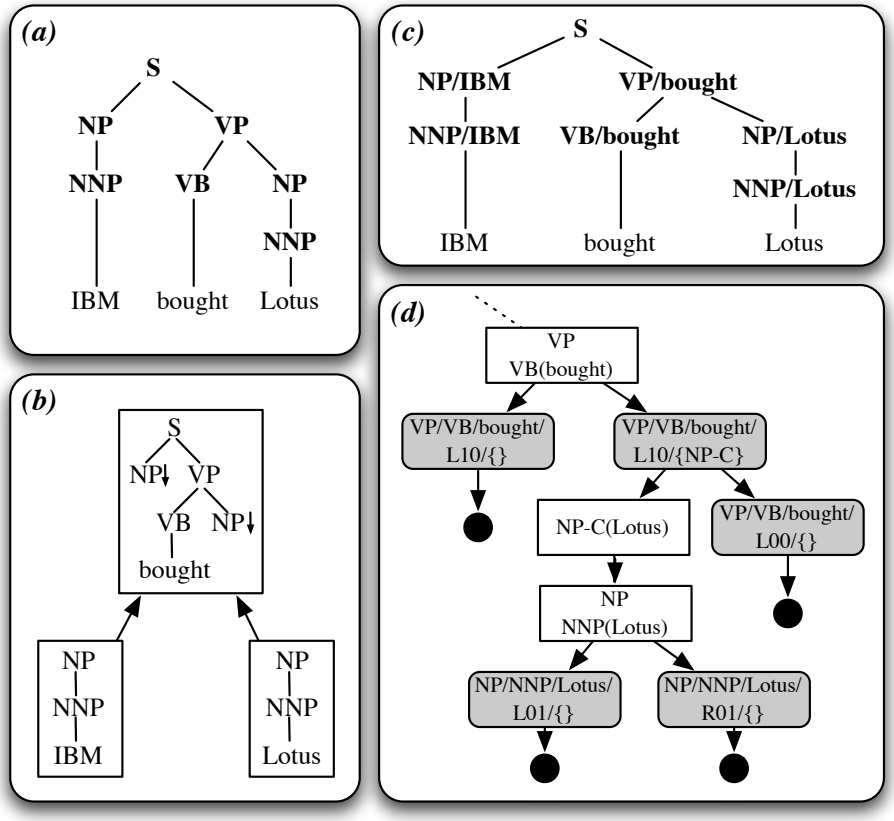


Figure 1.3: **Example Output Encodings: Parsing**. This figure shows four different encodings for the task of parsing a sentence. Structure *(a)* encodes the parse with a Treebank-style tree. Structure *(b)* encodes the parse with a TAG derivation tree [11]. Structure *(c)* encodes the parse with a lexicalized tree. Structure *(d)* encodes the parse in a structure reflecting the decomposition of Michael Collin's "Model 2" parser (see Figure 3.1 for a more detailed explanation of *(d)*).

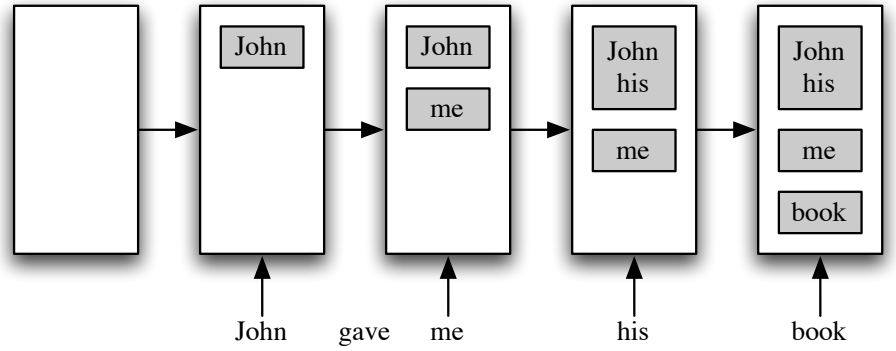


Figure 1.4: **Example Output Encoding: Coreference Resolution**. This figure shows a simplified example of an encoding for a coreference system, based loosely on Thomas Morton's coreference system [34]. This system maintains a "discourse model" (white box), consisting of a set of "entities" (shaded boxes). 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 proposal is on the effects of different problem decompositions; and 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," I 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 Structure of this Document

Chapter 2 provides the background for this proposal, including explanations of common techniques for decomposing a problem into sub-problems. Chapter 3 describes prior work on the effect of different output encodings, and transformations of output encodings, on the performance of supervised learning tasks. Chapter 4 introduces

a representation for output encodings in chunking problems; and describes several experiments that use a hill-climbing algorithm to explore the space of possible encodings. Chapter 5 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 6 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 7 describes the issues that arise when we use voting to combine models that use different output encodings; and describes an algorithm that can be used to overcome those issues. Chapter 8 presents a method for adding features to machine learning models that depend on non-local pieces of output structure. Chapter 9 proposes a set of research tasks to be completed for this dissertation.

Chapter 2

Background

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

- *Chunking*: find a set 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 [37, 44, 49]. 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 [37].

> *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; 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

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:

$$P(E|B) = P(E) \quad (2.2)$$

$$P(J|A, E, B) = P(J|A) \quad (2.3)$$

$$P(M|A, E, B, J) = P(M|A) \quad (2.4)$$

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.

---

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.

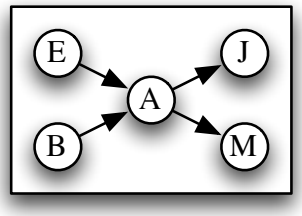


Figure 2.1: **Bayesian Network For the Alarm Problem**. Nodes in this graph represent variables: A indicates whether the alarm has gone off; E and B indicate whether there was an earthquake or burglary attempt respectively; and J and M indicate whether John or Mary respectively have called. Edges represent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

2.1.2 Decomposing Structured Output Values

In [7], 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 [7], 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 Chunking

A chunking task is any task that consists of finding some set 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.

	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 [43]. See Figure 1.2 for a graphical depiction of this example.

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` 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

[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.

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`.

2.2.2 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 [36] is an annotation of one million words of the Wall Street Journal portion of the Penn Treebank II [29] 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" [13]. 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.

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	$Y = Y_1, Y_2, ..., Y_n$
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 2.3: **Notation for Sequence Learning**.

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 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

[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 $\text{len}(\vec{x}) = \text{len}(\vec{y})$.

[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.

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 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:

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

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 2.4: **Notation for Hidden Markov Models**.

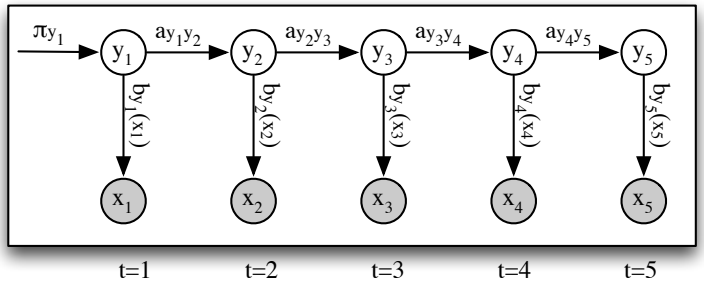


Figure 2.5: **HMM as a Bayesian Graphical Model**. This figure shows how HMMs are related to Bayesian Networks. Nodes are used to represent variables: the nodes marked y_t represent the states at each time step; and the nodes marked x_t represent the emitted symbols at each time step. Edges represent 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.

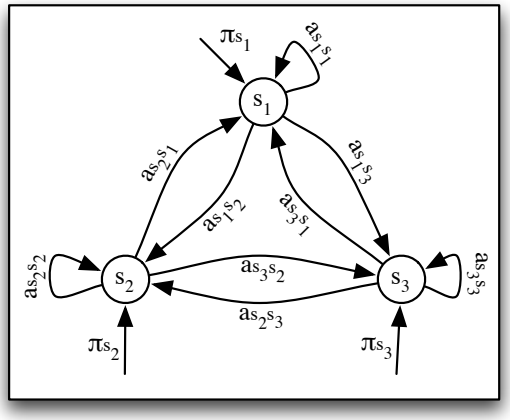


Figure 2.6: **HMM as a Finite State Machine**. This graphical depiction of an HMM highlights its relationship to finite state machines. This HMM has three states, s_1, s_2, and s_3. Arcs are labeled with probabilities: the arcs marked with π_i indicate that the HMM may start in any of the three states, with the given probabilities; and the arcs between states indicate the probability of transitioning between states. Symbol emission probabilities are not shown.

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:

$$\pi_s = \widehat{P}(y_1 = s) \tag{2.8}$$

$$= \frac{count(y_1 = s)}{N} \tag{2.9}$$

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 outgoing transitions from state s_i:

$$a_{s_i s_j} = \widehat{P}(y_t = s_i, y_{t+1} = s_j) \tag{2.10}$$

$$= \frac{count(y_t = s_i, y_{t+1} = s_j)}{count(y_t = s_i)} \tag{2.11}$$

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 CRF. 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:

$$v_s(1) = \pi_s b_s(x_1) \tag{2.13}$$

$$v_{s_i s_j}(t) = a_{s_i s_j} b_{s_j}(x_t) \qquad 2 \leq t \leq T \tag{2.14}$$

Using these edge weights, the probability of an output value $\vec{y}$ is simply the

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	$v_s(1)$	$=$	$\text{weight}\,(q_0 \rightarrow q_{1,s})$
	$v_{ss'}(t)$	$=$	$\text{weight}\,(q_{t-1,s} \rightarrow q_{t,s'})$
Max Forward Scores	$\delta_s(t)$		
Max Backward Scores	$\phi_s(t)$		

Figure 2.7: **Notation for Viterbi Graphs**.

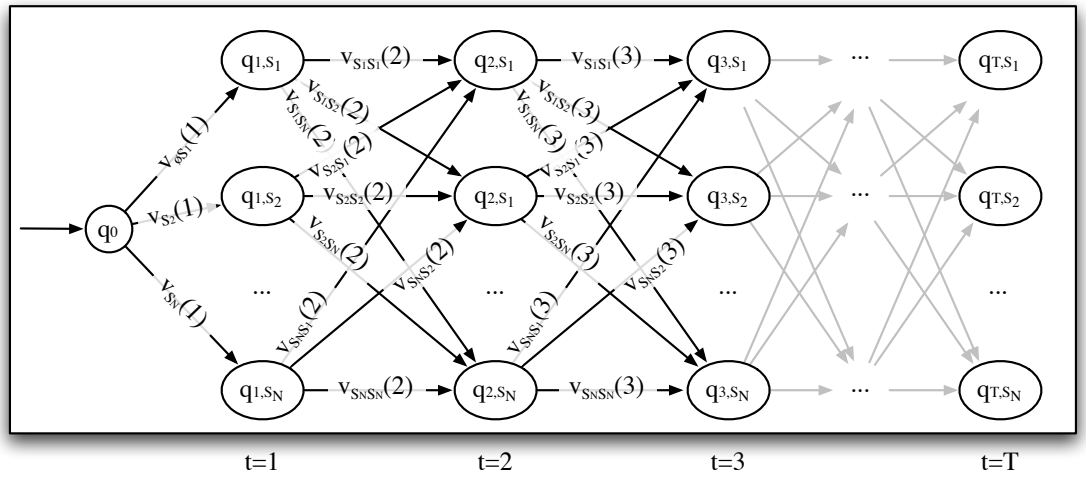


Figure 2.8: **Viterbi Graph**. This graphical structure is used for *decoding*, or finding the most likely output value, in HMMs, MEMMs, and linear chain CRFs. Each node q_{t,s_i} represents an assignment of a single output tag $y_t = s_i$. Edges are annotated with weights, such that the score of an 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.

product of the edge weights in the corresponding path:

$$P(\vec{y}, \vec{x}) = \pi_{y_1} \prod_{t=1}^{T-1} a_{y_t y_{t+1}} \prod_{t=1}^{T} b_{y_t}(x_t) \tag{2.15}$$

$$= (\pi_{y_1} b_{y_1}(x_1)) \left(\prod_{t=2}^{T} a_{y_{t-1} y_t} b_{y_t}(x_t) \right) \tag{2.16}$$

$$= v_{y_1}(1) \prod_{t=2}^{T} v_{y_{t-1} y_t}(t) \tag{2.17}$$

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]:

$$\delta_s(1) = v_s(1) \tag{2.18}$$

$$\delta_s(t) = max_{s'} \delta_{t-1}(s') v_{s's}(t) \qquad 1 < t \leq T \tag{2.19}$$

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 $\delta_s(t)$:

$$y_T^* = \arg\max_s \delta_s(T) \tag{2.20}$$

$$y_t^* = \arg\max_s \delta_t(s) v_{s \vec{y}_{t+1}^*}(t) \qquad 1 \leq t < T \tag{2.21}$$

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.

$$\phi_s(T) = 1 \tag{2.22}$$

$$\phi_s(t) = max_{s'} v_{ss'}(t+1) \phi_{t+1}(s') \qquad 1 \leq t < T \tag{2.23}$$

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 [31].

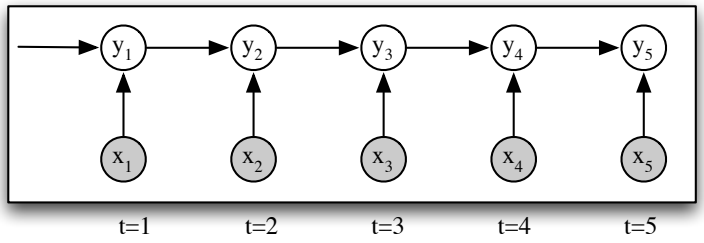


Figure 2.9: **MEMM as a Bayesian Graphical Model**. This figure shows how MEMMs are related to Bayesian Networks. Nodes are used to represent variables: the nodes marked y_t represent the states at each time step; and the nodes marked x_t represent the emitted symbols at each time step. Edges represent 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

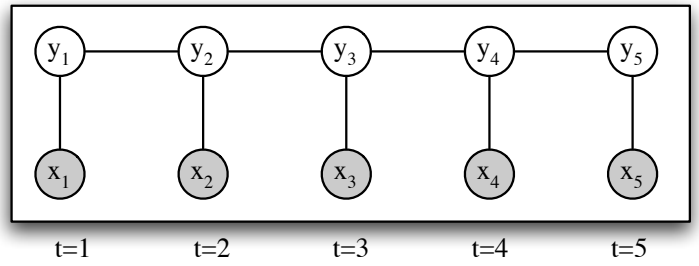


Figure 2.10: **Linear Chain CRF as a Bayesian Graphical Model**. This figure shows how Linear Chain CRFs are related to Bayesian Networks. Nodes are used to represent variables: the nodes marked y_t represent the states at each time step; and the nodes marked x_t represent the emitted symbols at each 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 [51]. These methods all attempt find the set of weights that maximize the log-likelihood of a given training corpus $(\vec{x}_k, \vec{y})_{k\,k=1}^{N}$:

$$\lambda* = \arg\min_{\lambda} \left(\sum_k p_\lambda(\vec{y}_k|\vec{x}_k) \right) \tag{2.30}$$

$$= \arg\min_{\lambda} \left(\sum_k [\lambda \cdot F(\vec{y}_k, \vec{x}_k) - logZ_\lambda(\vec{x}_k)] \right) \tag{2.31}$$

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 input. For Linear chain CRFs, we set the Viterbi graph edge weights as follows:

$$v_s(1) = exp\left(\sum_{a \in A} \lambda_a f_a(\vec{x}, y_1, 1) \right) \tag{2.32}$$

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

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 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're 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 $Z(\vec{x})$:

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

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 system 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, $elements(\vec{y}^{(i)})$.

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

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

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:

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

$$\tag{2.43}$$

Chapter 3

Prior Work

3.1 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.

3.1.1 Error Correcting Output Codes

One such attempt is Error Correcting Output Codes (Dietterich & Bakiri, 1995), which decomposes a single classification task into a set of subtasks that are implemented by base learners [10]. 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 & 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.

3.1.2 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. [14, 20]

3.1.3 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. [1, 9]

3.2 Output Encodings for Structured Output Tasks

3.2.1 Chunking Representations

The Noun Phrase chunking task was originally formulated as a sequence tagging task by Ramshaw & Marcus [43]. 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 performed by Sang & Veenstra. In [53], Sang & Veenstra adapted a memory-learning NP chunker to use seven different output encodings, including four of the five encodings described in Section 2.2.1 (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 Sang (2000) [45], 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 [8]. 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 & Matsumoto (2001) [26] 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 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 & Sarkar [48] also built a voting system based on the five encodings defined by 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.

3.2.2 Semantic Role Representations

Traditionally, the arguments of a verb have been labelled using *thematic roles*, which were first introduced in the mid 1960s [18, 15, 21]. 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 [12]. 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 [13]. 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 [36]. 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 5, 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 [6, 16, 19, 35, 56, 41, 42, 54]. For two years, the CoNLL

workshop has made this problem the shared task [3, 4]. Most existing systems use a series of independent classifiers. For example, many systems break the Semantic Role Labeling task into two sub-tasks, 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 [17, 39, 52, 50, 54].

Transforming SRL Representations

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

3.2.3 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 [22]. 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 [7]. 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 3.1 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.[1]

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 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.

[1]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 [7] and [2] for a more detailed account.

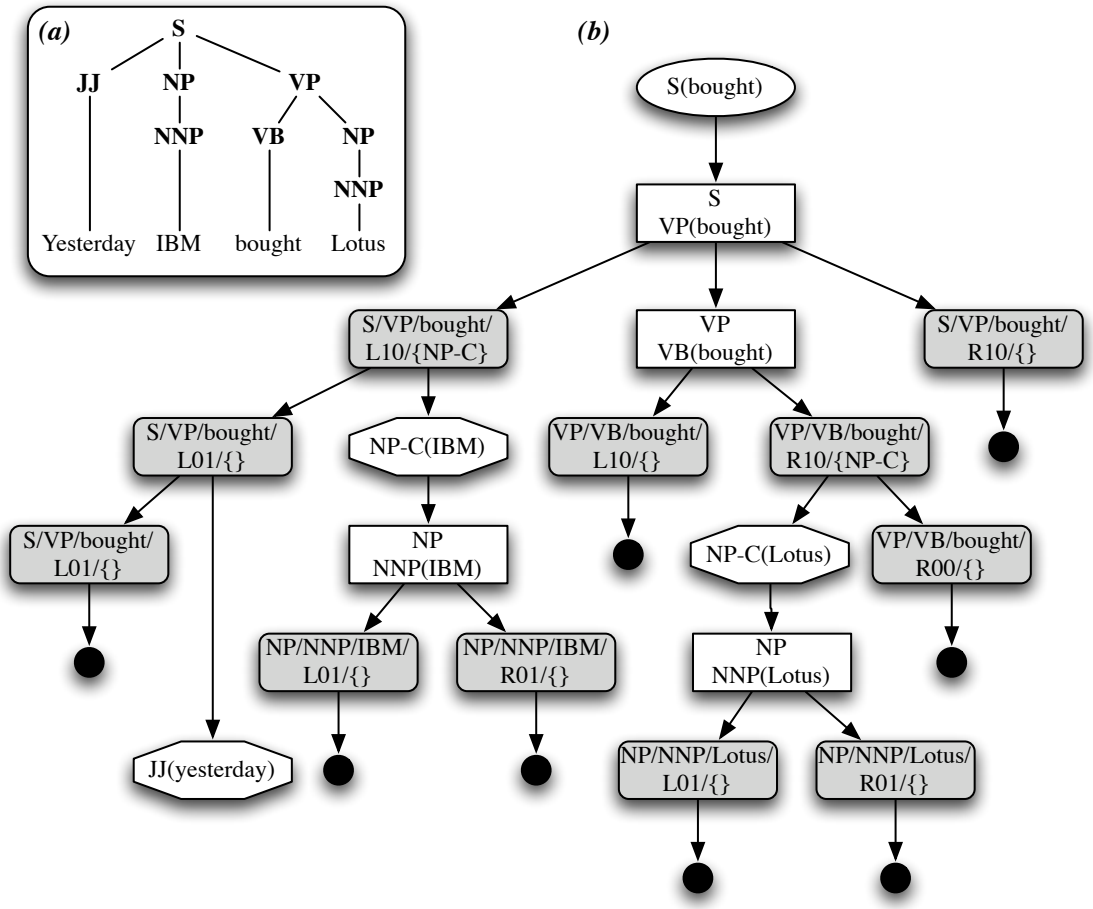


Figure 3.1: **Collins' "Model 2" Parser as Tree Transformation**. This figure illustrates how Collins' Parser can be modelled using the notion of encoding transformation. The structure *(a)* shows the canonical parse tree encoding for a simple example sentence. The structure *(b)* shows an encoding of 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 corespond 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 [7, 2] for more information about Collins' parser.

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 [2]. 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 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 [23]. 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 [25, 24]. 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 model the probability of a tree T as:

$$P(T) = P\left(\tau_{PCFG}(T)\right) P\left(\tau_{dep}(T)\right) \tag{3.1}$$

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 3.2 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

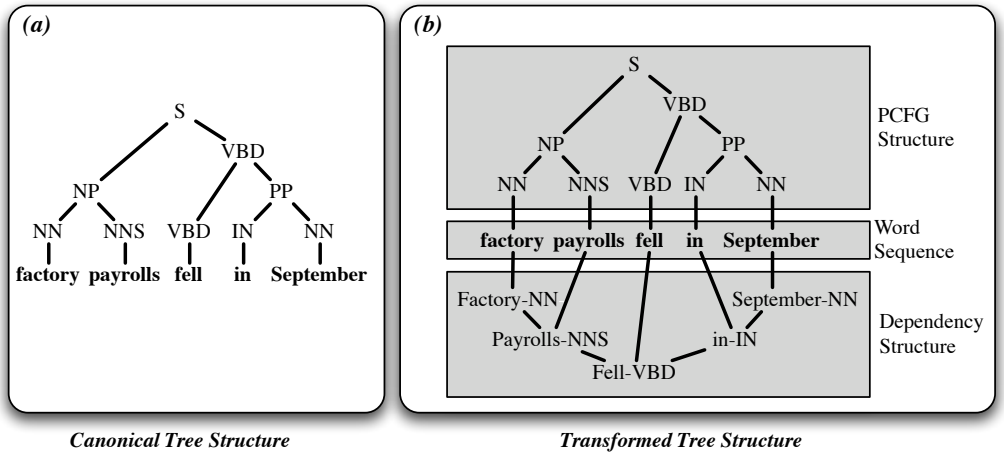


Figure 3.2: **Klein & Manning's Factored Model as a Tree Transformation**. Klein and Manning's factored parsing model $P(T, D) = P(T)P(D)$ can be thought of as a tree transformation that replaces the canonical structure *(a)* with a new structure *(b)* that describes the sentence's structure using two separate (disconnected) pieces: one describing the sentence's PCFG structure, and the other describing its dependency structure.

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 3.1 could be replaced by the following conditional variant, where S is the input sentence:

$$P(T|S) = P\left(\tau_{PCFG}(T|S)\right) P\left(\tau_{dep}(T|S)\right) \tag{3.2}$$

Since both models are conditioned on S, they can no longer generate incompatible terminal strings.[2]

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

[2]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.

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:

$$P_{outside}(\alpha) = \max_{T:\alpha \in T} P(T) - P(\alpha) \tag{3.3}$$

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:

$$P_{outside}(\alpha) \leq P_{outside}\left(\tau_{PCFG}(\alpha)\right) P_{outside}\left(\tau_{dep}(\alpha)\right) \tag{3.4}$$

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.

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 [30]. 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, Barrett, Thibaux, & Klein [38] use 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 4

A Hill Climbing Algorithm for Optimizing Chunk Encodings

Chunking is the task of finding a set of non-overlapping sub-sequences in a given sequence. In this chapter, we will explore how the use of different tag-based encodings for chunk outputs affects the ability of sequential learning models to learn chunking tasks. Section 4.1 shows how finite state transducers can be used to represent tag-based encodings of chunk outputs, and discusses the properties that such transducers must have. Section 4.2 defines various transformations that can be used to modify existing encodings, and Sections 4.3 and 4.4 show how those transformations can be used to improve the encoding used for chunking.

4.1 Representing Chunk Encodings with FSTs

As was discussed in Chapter 1, output encodings can be defined using reversible transformations with respect to a chosen canonical encoding. In the case of chunking, we will use finite state transducers (FSTs) to represent encodings. We will (somewhat arbitrarily) choose `IOB1` as the canonical encoding.[1] Given an encoding's FST, we

[1]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

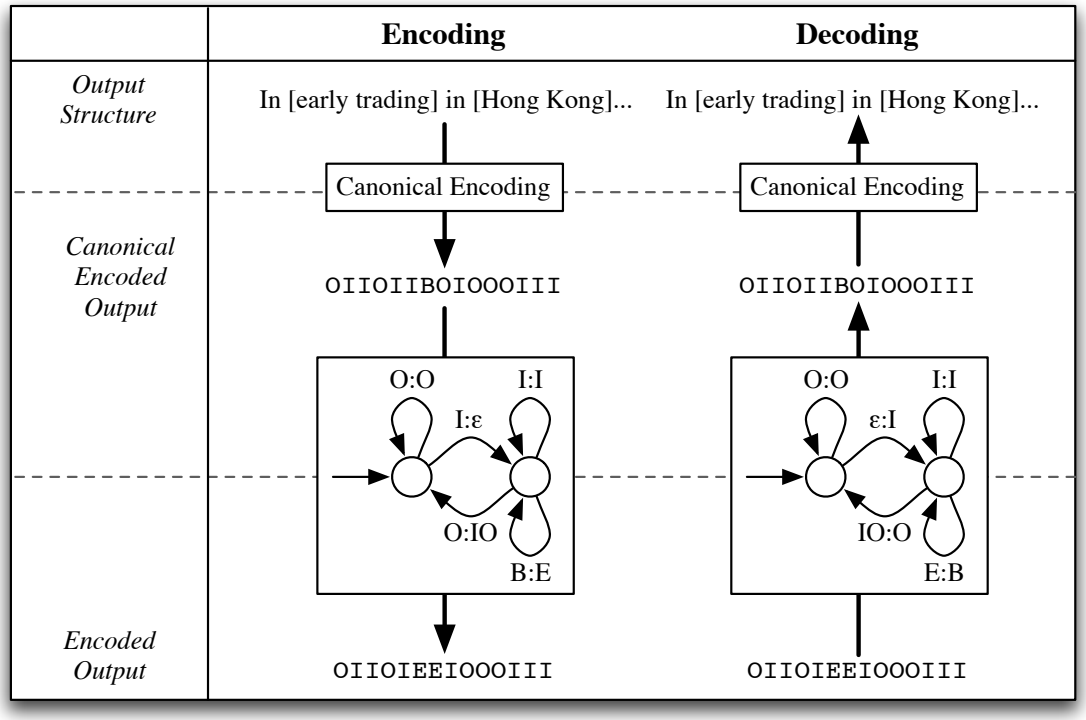


Figure 4.1: **Encoding Chunk Sequences with FSTs**. Chunk structure encodings are represented using FSTs. On the left, the FST for the `IOE1` encoding is used to encode a chunk sequence, by first generating the canonical (`IOB1`) encoding, and then translating that encoding with the FST. On the right, that `IOE1` encoding is decoded back to a chunk structure by first applying the reversed `IOE1` FST; and then interpreting the resulting string using `IOB1`.

can encode a chunking into a tag sequence by first encoding the chunking using `IOB1`; and then applying the FST to the `IOB1` tag sequence. To decode a tag sequence, we first apply the reversed FST to the tag sequence, to get an `IOB1` tag sequence; and then decode that tag sequence into a chunking, using `IOB1`. An example of the encoding and decoding procedure is illustrated in Figure 4.1.

encoding does not prevent us from expressing any encodings that a different canonical encoding would allow.

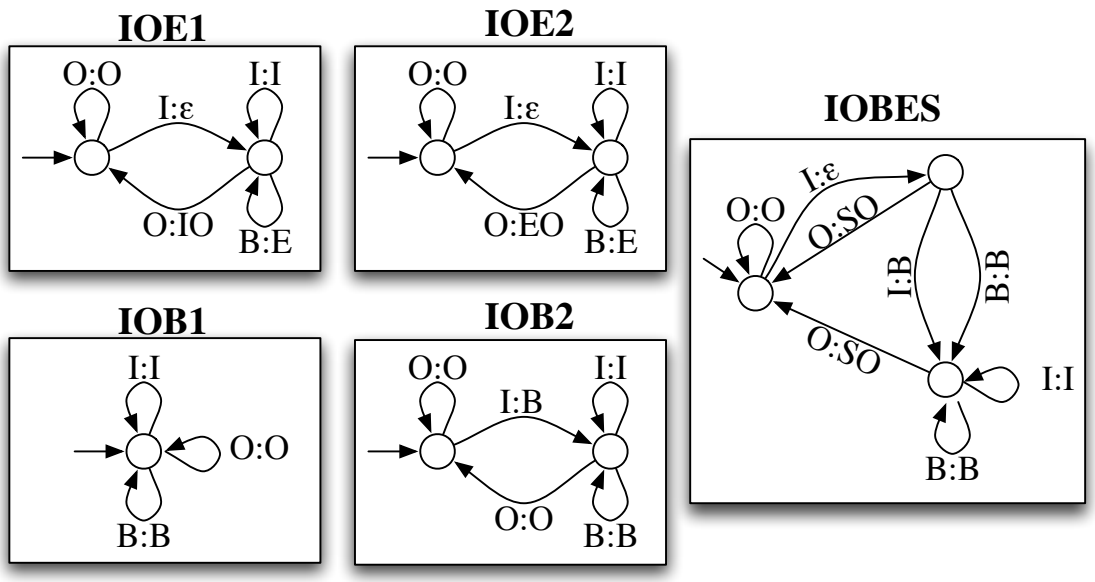


Figure 4.2: **FSTs for Five Common Chunk Encodings**. Each transducer takes an `IOB1`-encoded string for a given output value, and generates the corresponding string for the same output value, using a new encoding. Note that the `IOB1` FST is simply an identity transducer; and note that the transducers 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.

4.1.1 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.

4.1.2 Necessary Properties for Representing Encodings with FSTs

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

1. The FST's inverse should be deterministic.[2] Otherwise, we will be unable to choose a single unique output for some combination of base decision outcomes.
2. The FST should recognize exactly the set of valid input strings.
   - If it does not recognize some valid input string, then there is no way to map that input to the new encoding.
   - If it recognizes some invalid input string, then there exists some output string that maps back to that invalid input.

[2] Or at least determinizable.

3. The FST should always generate an output string with a length equal to its input string's length. Otherwise, it will not be possible to align the base decisions with corresponding tokens.

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 input.

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 (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 $3^n - 3^{n-1} - 3^{n-2}$; and there is no integer k such that $k^n = 3^n - 3^{n-1} - 3^{n-2}$.[3]

We must therefore relax at least one of these two properties. Relaxing the first property (deterministic FSTs) will make training harder; and relaxing the second property (complete FSTs) will make testing harder. For the purposes of this dissertation, we will choose to relax the second property.

4.1.3 Inverting the Output Encoding

Recall that the motivation behind this second property is that we need a way to map *any* set of annotation elements generated by the machine learning system back to a corresponding structured output value. As an alternative to requiring that the

[3]To see why the number of possible chunkings is $3^n - 3^{n-1} - 3^{n-2}$, 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 3^{n-2}).

FST generate every output string, we can define a method for building output values from annotation elements, even if those annotation elements do not form a "valid" annotation. One such method is to assume that one or more of the annotation elements was chosen incorrectly; and to make the minimal set of changes to the annotation elements such that the annotation becomes valid.

In order to compute the optimal output value corresponding to each set of annotation elements, we can use 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.

3. Assign a weight of zero to all arcs in the FST.

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.[4]

5. Determinize the resulting FST, using a variant of the algorithm presented in (Mohri, 1997) [33].

The resulting FST will accept all sequences of annotation elements, and will generate for each sequence the output value that is generated with the fewest number of changes to the given annotation elements.

4.1.4 FST Characteristics

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

[4]Alternatively, the weights could be decided based on some measure of the confusability of x and y.

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

Theorem 4.1. *In a chunk 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 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* output offset(S)*, such that on any path p from the initial state to S, $|in(p)| - |out(p)| =$* 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, output offset(S) is unique. □

4.2 Chunk-Encoding FST Modifications

The previous section showed how FSTs can be used to represent output encodings for chunking tasks. This section turns to the questions of how different encodings are related to one another, and of how we can improve upon an existing encoding scheme. In other words, given the space of chunk-encoding FSTs, we want to know how different elements of the space relate to one another; and we want to define a search algorithm for finding the elements in this space that maximize performance.

The basic tool we will use to examine these questions is *FST modifications*. In particular, we will define a set of modifications that can be used to transform any chunk-encoding FST into any other chunk-encoding FST. We can then express the relationship between any two encodings by examining what modifications would need to be made to the first FST to turn it into the second FST. Furthermore, this set of modifications defines a topology on the space of FSTs, which we can use to search for chunk-encodings that improve chunking performance. The remainder of this section describes the set of FST modification operations that I have defined for modifying chunk-encodings. Taken together, these modification operations are sufficient to generate any chunk-encoding FST.

4.2.1 State Splitting

The state splitting operation is used to introduce new structure to the chunk-encoding FST, by increasing the number of states it contains. This operation 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 chunk-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 $\langle S \rightarrow Q[x:y] \rangle$, add a new arc $\langle S' \rightarrow Q[x:y] \rangle$. 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').

The state splitting operation is a very general operation, and can be used to make relatively drastic changes to the transduction. As a result, this operation is too general to allow for efficient search for improved chunk-encodings. I therefore define several specializations of this operation, which make smaller modifications that are more amenable to search: arc specialization and loop unrolling .

Arc Specialization

The arc specialization operation is a special case of state splitting . It acts on a single arc e with destination state S, and makes the following changes to the FST:

1. Create a new state S'.
2. For each outgoing arc $e_1 = \langle S \rightarrow Q[\alpha:\beta] \rangle$, $S \neq Q$, add an arc $e_2 = \langle S' \rightarrow Q[\alpha:\beta] \rangle$.
3. For each loop arc $e_1 = \langle S \rightarrow S[\alpha:\beta] \rangle$, add an arc $e_2 = \langle S' \rightarrow S'[\alpha:\beta] \rangle$.

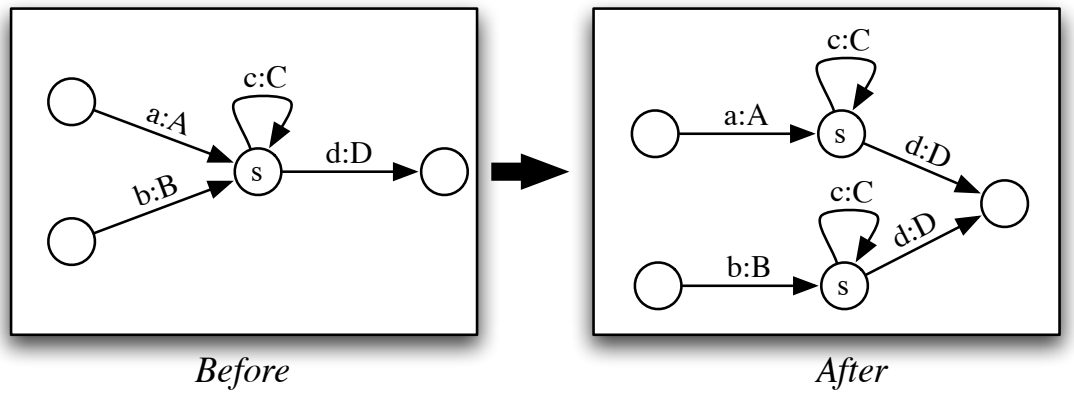


Figure 4.3: **Arc Specialization**.

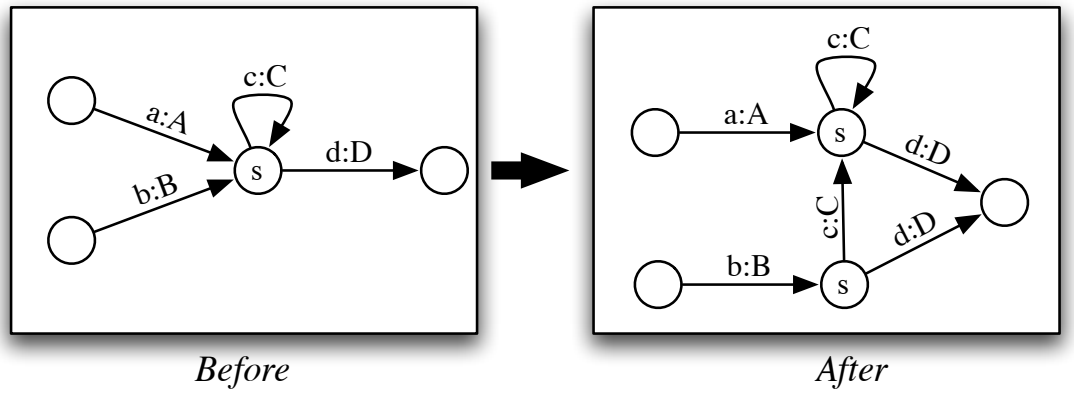


Figure 4.4: **Arc Specialization Variant**.

4. Change arc e's destination from S to S'.

An example of this operation is shown in figure 4.3. Like all state splitting operations, arc specialization does not directly modify the transduction performed by the FST. However, it does provide the FST with two states that can be used to distinguish the path that was taken to arrive at the original S state.

In a variant transformation, shown in figure 4.4, the loop arcs from S can be replaced by arcs from $S \rightarrow S'$, rather than loop arcs in S'. I.e., this variant operation makes the following changes to the FST. (This operation differs from the basic arc

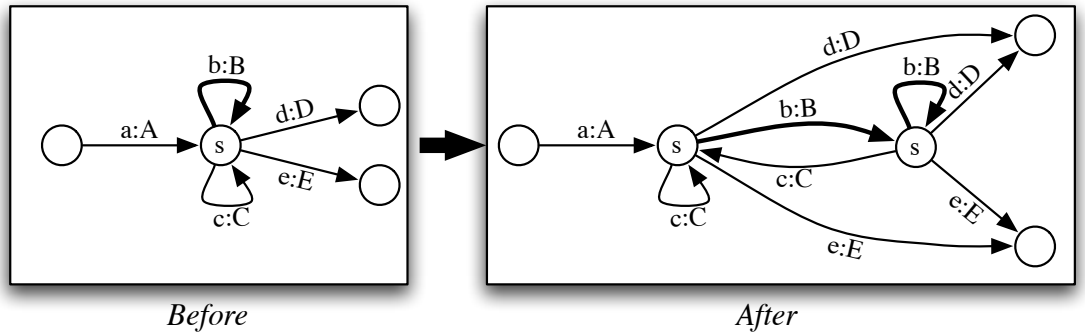


Figure 4.5: **Loop Unrolling**.

specialization operation in step 3.)

1. Create a new state S'.
2. For each outgoing arc $e_1 = \langle S \rightarrow Q[\alpha : \beta] \rangle$, $S \neq Q$, add an arc $e_2 = \langle S' \rightarrow Q[\alpha : \beta] \rangle$.
3. For each loop arc $e_1 = \langle S \rightarrow S[\alpha : \beta] \rangle$, add an arc $e_2 = \langle S' \rightarrow S[\alpha : \beta] \rangle$.
4. Change the destination of arc e from S to S'.

Loop Unrolling

The loop unrolling operation is another special case of state splitting. It 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 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.
3. Change the destination of loop arc e from S to S'.

Loop unrolling is similar to arc specialization , except that the new copy of the arc being unrolled has its destination set to S', while all other copies of loop arcs

have their destinations set to S. Figure 4.5 shows the loop unrolling operation. The arc being unrolled is marked in bold.

Like any state splitting operation, loop unrolling does not actually change the transduction performed by the FST. However, it does provide the FST with two states that can be used to distinguish the first pass through the loop from subsequent passes through the loop.

4.2.2 Output Relabeling

The output relabeling operation replaces the output strings on a state S's incoming and outgoing arc 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 output offset(S) remains unique for the state; the new value for output offset(S) will differ from the original value for output offset(S) by n.

Like the state splitting operation, output relabeling is a very general operation, and can be used to make relatively drastic changes to the transduction. I therefore define two specializations of this operation, which are more amenable to search:

- 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 symbol that is not used anywhere else in the transducer.

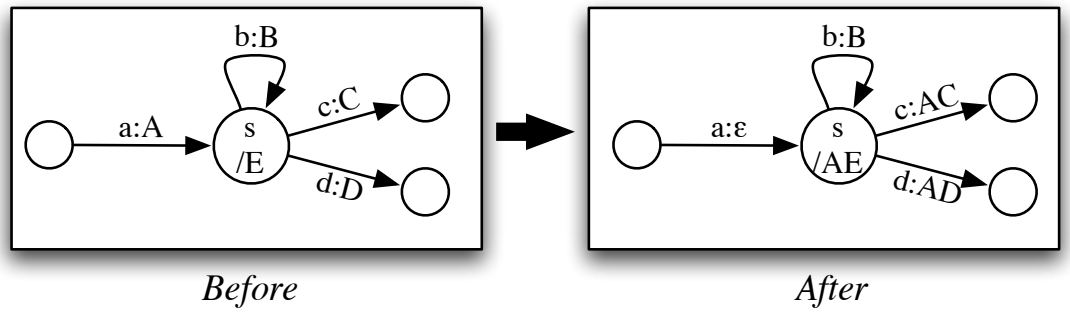


Figure 4.6: **Output Delay**.

- 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 distinct from all other output strings from state S, but is composed of output symbols that are already used in the transducer. The length of the new output string must equal the length of the old output string, to ensure a consistent output offset(S) value.

4.2.3 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.
- 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 4.6 shows an example of the output delay operation.

4.2.4 Feature Specialization

The final FST modification operation, feature specialization , differs from the other operations in that it makes use of features of the *input value*. For example, in a

noun phrase chunker, this operation can be used to split a state in two, depending on whether the current *input* word is a preposition or not. The feature specialization operation acts on an arc $a = \langle S \rightarrow Q[\alpha : \beta x \gamma] \rangle$, an input feature f, and a feature value v. It makes the following changes to the FST:

- Duplicate the destination state Q. In other words, create a new state, Q', which has the same outgoing edges as Q.

- Replace arc a with two new arcs: $\langle S \rightarrow Q[\alpha : \beta x_{[f=v]} \gamma] \rangle$ and $\langle S \rightarrow Q[\alpha : \beta x_{[f \neq v]} \gamma] \rangle$.

Where $x_{[f=v]}$ is an input symbol that will only match if the corresponding token's tag is x and the corresponding token's feature f has value v; and $x_{[f \neq v]}$ is an input symbol that will only match if the corresponding token's tag is x and the corresponding token's feature f does not have value v.

4.3 A Hill Climbing Algorithm for Optimizing Chunk Encodings

Having defined a set of modification operations for chunk-encoding FSTs, we can now use those operations to search for improved chunk 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 help avoid sub-optimal local maxima, we can use a fixed-size beam search. In order to test the effectiveness of this approach, I applied the hill-climbing procedure shown in Figure 4.7 to the task of NP chunking.

1. Initialize `candidates` to be the singleton set containing the identity transducer.

2. Repeat ...

   (a) Generate five new FSTs, by randomly applying modification operations to members of the `candidates` set.
   (b) Evaluate each of these new FSTs, by training them on the training set and testing them on held-out data.
   (c) Add the new FSTs to the `candidates` set.
   (d) Sort the `candidates` set by their score on the held-out data, and discard all but the ten highest-scoring candidates.

   ... until no improvement is made for three consecutive iterations.

3. Return the candidate FST with the highest score.

Figure 4.7: **A Hill Climbing Algorithm for Optimizing Chunk Encodings**.

Feature	Description
word.lower	The lower-case version of the token.
form	A string describing the capitalization and form of the token: one of *number*, *allcaps*, *capitalized*, *lowercase*, or *other*.
sentpos	Position in the sentence: one of *start*, *mid*, *end*.
word.pos	The part of speech tag of the token.
prev.pos	The part of speech tag of the previous token.
next.pos	The part of speech tag of the next token.

Figure 4.8: **Feature Set for the HMM NP Chunker**.

4.3.1 Experiments

In my initial experiments, I used a Hidden Markov Model as the underlying machine learning algorithm for learning tag sequences. The features used are listed in Figure 4.8. Training and testing were performed using the noun phrase chunking corpus described in Ramshaw & Marcus (1995) [43]. A randomly selected 10% of the originally training corpus was used as held-out data, to provide feedback to the hill-climbing system.

The baseline system, which used the canonical encoding (`IOB1`), achieves an F-score of 87.6%. Using the hill-climbing algorithm to find an improved encoding, and

training and testing with that new encoding, increases the system's F-score to 89.2%.

It should be noted that both of these F-scores are significantly below the current state of the art for NP chunking. This is probably a result of the fact that the underlying HMM model is not sufficiently flexible to learn these tagging sequences. In particular, HMM models assume that all input features are mutually independent; but clearly, the feature set I used included a number of dependant features.

I therefore conducted a second experiment, in which I replicated the state-of-the-art NP chunker described in Sha & Pereira (2003) [47], using the Mallet software package [32]. This system uses a Linear Chain Conditional Random Field to model the tag sequence. CRF labels y_i are defined to be pairs of output tags $c_{i-1}c_i$. The features used are listed in Figure 4.9. Using this state-of-the-art NP chunker as my underlying system, the hill-climbing procedure did not lead to any significant change in the performance of the chunker.

There are several possible reasons for this. First, the performance of Sha & Pereira's NP chunker is already very close to human performance scores. In other words, there is very little headroom for making improvements, since we can not reasonably expect a supervised system to perform better than humans. Second, it is possible that there exist superior encodings for use with Sha & Pereira's NP chunker, but that the set of local FST modifications that are available to the hill climbing algorithm do not provide a sufficiently monotonic gradient for the algorithm to find such encodings. The use of a different set of FST modification operations might allow the hill climbing algorithm to find encodings that the current system is unable to find.

4.4 Optimizing the Hill-Climbing Search

Since the hill-climbing procedure must evaluate every candidate output encoding on the held-out data, it can be relatively slow. Furthermore, the search is currently

Feature	Description
$c(y_i)$	The current output tag.
$y_i = y$	The current state (current and previous output tag).
$y_i = y, y_{i-1} = y'$	A tuple of the current state and the previous state (current and previous two output tags).
y_i, w_{i+n}	A tuple of the current state and the $i+n$th word, $-2 \leq n \leq 2$.
y_i, w_i, w_{i-1}	A tuple of the current state, the current word, and the previous word.
y_i, w_i, w_{i+1}	A tuple of the current state, the current word, and the next word.
y_i, t_{i+n}	A tuple of the current state 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 state and the two consecutive part of speech tags starting at word $i+n$, $-2 \leq n \leq 1$.
$y_{i+n-1}, t_{i+n}, t_{i+n+1}$	A tuple of the current state, and three consecutive part of speech tags centered on word $i+n$, $-1 \leq n \leq 1$.
$c(y_i), w_{i+n}$	A tuple of the current output tag and the $i+n$th word, $-2 \leq n \leq 2$.
$c(y_i), w_i, w_{i-1}$	A tuple of the current output tag, the current word, and the previous word.
$c(y_i), w_i, w_{i+1}$	A tuple of the current output tag, the current word, and the next word.
$c(y_i), t_{i+n}$	A tuple of the current output tag and the part of speech tag of the $i+n$th word, $-2 \leq n \leq 2$.
$c(y_i), t_{i+n}, t_{i+n+1}$	A tuple of the current output tag and the two consecutive part of speech tags starting at word $i+n$, $-2 \leq n \leq 1$.
$y_{i+n-1}, t_{i+n}, t_{i+n+1}$	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 4.9: **Feature Set for the CRF NP Chunker**.

fairly undirected: it simply chooses modification operations at random. This section discusses three approaches that might be used to increase the speed and accuracy of the hill-climbing algorithm.

4.4.1 Predicting Which Modifications Will Help

The first approach attempts to provide some guidance to the hill-climbing search by predicting which modification operations will improve performance. In particular, there are certain conditions under which we might expect a given modification operation to yield more positive results than others.

For example, consider the operation of feature specialization , which splits a given FST state based on a feature of the input value. Essentially, this transformation is allowing us to distinguish two pieces of output structure that were previously grouped into the same equivalence class. In other words, this transformation divides one group of equivalent sub-problems into two sub-groups. If these two sub-groups behave differently from each other, then this transformation will allow the underlying model to capture those differences. But at the same time, this split will increase the data sparsity for each of the new sub-groups. Thus, we expect the feature specialization operation to be more likely to improve performance if data sparsity is not an issue. In particular, we should be more willing to split a symbol if we have more training data for that symbol.

We can test this hypothesis by performing the feature specialization operation on a variety of features, and looking at the relationship between the data support for the feature and the resulting change in performance. Based on this experiment, we can come to conclusions about how much data support should be required before we are willing to apply the feature specialization operation. We can then apply those results back to the hill-climbing system, by increasing the chance that it will apply the feature specialization operation when there is sufficient data support, and decreasing the chance when there is not.

4.4.2 Problem-Driven Learning

A second approach that could provide guidance to the hill-climbing algorithm would be to examine the specific problems that are made by a given encoding, and then use those problems to focus the search area of the hill-climbing algorithm. Given the fact that an encoding makes a large number of mistakes on a given output tag, we could examine the context of that output tag, and attempt to augment the output encoding in ways that would allow the system to improve performance. For example, we could search for features of the input, both local and non-local, which might allow the system to distinguish the incorrect tags from the correct ones; and use those features to apply the feature specialization operation.

4.4.3 Searching for Patterns

Often, a given chunking task will include several special classes of output value that act differently from the rest. For example, in the task of noun phrase chunking, certain noun phrase types, such as date expressions, money expressions, and proper nouns, act very differently from other noun phrases. In such cases, it may be beneficial to split these special classes out, by tagging them with unique output tags.

In order to find such special classes automatically, we can use unsupervised techniques to search through the training corpus (including both input and output values) for instances that act differently from the rest. For example, clustering techniques could be used to find classes of instances that are significantly different from each other. These classes could then be specialized in the output encoding, by assigning different output tags depending on which class each instance belongs to.

Alternatively, if we have prior domain knowledge about such special classes, we can let the hill climbing know about them, giving it the option of using specialized output tags to encode them.

4.5 Proposed Work

For my dissertation, I plan to refine the use of hill-climbing algorithms to find improved output encodings for chunking tasks. In particular, I plan to carry out the following experiments:

- Evaluate the performance of the current hill-climbing system on a chunking task with more headroom than Base NP Chunking, such as Biomedical Named Entity Detection.
- Perform several experiments that explore factors influencing the impact of existing modification operations on system performance, and use those experiments to provide improved guidance to the hill-climbing system.
- Apply a problem-driven learning approach to provide improved guidance to the hill-climbing system.
- Evaluate the effect of specializing chunk-encoding output tags by dividing instances based on unsupervised clustering algorithms.

Chapter 5

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 (PropBank) corpus [36] avoids this issue by using theory-agnostic labels (Arg0, Arg1, ..., Arg5), and by defining those labels to have verb-specific meanings. 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 encoded 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 [41, 40] 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 PropBank additional genres, 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.

5.1 VerbNet

VerbNet [46] consists of hierarchically arranged verb classes, inspired by and extended from classes of Levin 1993 [27]. 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.

5.2 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 [28], 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 [28].

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

5.3 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 5.1 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.

5.4 Training a 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 5.1: **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.

5.4.1 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, and to expand the size and variety of the training data. 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.

5.5 SRL Experiments on Linked Lexical Resources

This section describes joint work done with Szu-ting Yi and Martha Palmer.

In [28] and [55], we performed several preliminary experiments to verify the feasibility of performing semantic role labeling with VerbNet thematic roles. In these experiments, 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.

5.5.1 The 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 [56, 57].

5.5.2 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.

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 5.1: **Thematic Role Grouping A**. This grouping of thematic roles was used for subdividing Arg1 in Experiment 5.5.2. Karin Kipper assisted in creating the groupings.

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

Figure 5.2: **Thematic Role Grouping B**. This grouping of thematic roles was used for subdividing Arg2 in Experiment 5.5.2; and for mapping Args2-5 in Experiment 5.5.4. Karin Kipper assisted in creating the groupings.

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 grouped the VerbNet thematic roles into coherent groups of similar thematic roles, shown in Figure 5.1 (for the Arg1 experiments) and Figure 5.2 (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}.

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 $\text{Arg}n_{\text{group}i}$ were simply treated as members

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 5.2: **Results from Experiment 5.5.2 (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 5.3: **Results from Experiment 5.5.2 (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.

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 5.2 gives the results of the mapping on SRL overall performance, tested on the WSJ corpus Section 23; Table 5.3 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 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 5.2 and Table 5.3 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.

5.5.3 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 5.4, we discover the following:

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 5.4: **Confusion Matrix for Experiment 5.5.2**. Confusion matrix on the 1,539 instances which ARG2-Mapped tags correctly and ARG2-Original fails to predict.

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.

5.5.4 Applying the SemLink Mapping to Multiple Arguments

We also performed an experiment to evaluate the effect of applying the SemLink mapping to multiple arguments. Since PropBank arguments Arg0 and Arg1 are already quite coherent, we left them as-is in the new label set. But since arguments Arg2-Arg5 are highly overloaded, we replaced them by mapping them to their corresponding VerbNet thematic role.

In order to prevent the training data from these mapped labels from becoming too sparse (which would impair system performance) we grouped the VerbNet thematic roles into five coherent groups of similar thematic roles, shown in Figure 5.2. Our

System	Precision	Recall	F_1
Original	90.65	85.43	87.97
Mapped	88.85	84.56	86.65

Table 5.5: **Results from Experiment 5.5.4 (overall)**. Overall SRL System performance using the PropBank tag set ("Original") and the augmented tag set ("Mapped"). *Note that these results are not directly comparable, since the these two tag sets have different information content.*

new tag set therefore included the following tags: **Arg0** (*agent*); **Arg1** (*patient*); **Group1** (*goal*); **Group2** (*extent*); **Group3** (*predicate/attrib*); **Group4** (*product*); and **Group5** (*instrument/cause*).

Training our SRL system using these thematic role groups, we obtained performance similar to the original SRL system. However, *it is important to note that these performance figures are not directly comparable*, since the two systems are performing different tasks. In particular, the role labels generated by the original system are verb-specific, while the role labels generated by the new system are verb-dependent.

Results

For our testing and training, we used the portion of Penn Treebank II that is covered by the mapping, and where at least one of Arg2-5 is used. Training was performed using sections 2-21 of the Treebank (10,783 instances); and testing was performed on section 23 (859 instances). Table 5.5 displays the performance score for the SRL system using the augmented tag set ("Mapped"). The performance score of the original system ("Original") is also listed, for reference; however, as was discussed above, these results are not directly comparable because the two systems are performing different tasks.

The results indicate that the performance drops when we train on the new argument labels, especially on precision when we evaluate the systems on only Arg2-5/Group1-5 (see Table 5.6). However, it is premature to conclude that there is no benefit from the VerbNet thematic role labels. Firstly, we have a very few mapped

System	Precision	Recall	F_1
Original	97.60	83.67	90.10
Mapped	91.70	82.86	87.06

Table 5.6: **Results from Experiment 5.5.4 (Arg2-Arg5 only)**. SRL System performance evaluated on only Arg2-5 (Original) or Group1-5 (Mapped). *Note that these results are not directly comparable, since the these two tag sets have different information content.*

Arg3-5 instances (less than 1,000 instances); secondly, we lack test data generated from a genre other than WSJ to allow us to evaluate the robustness (generality) of SRL trained on the new argument labels.

5.6 Proposed Work

In order to further explore the hypothesis that mapping PropBank semantic role labels to a more coherent set of role labels can improve both the performance and the ability to generalize for SRL systems, I plan to perform the following experiments:

- First, I will repeat the experiments described in Section 5.5.2 on the remaining arguments (Arg0 and Arg3-Arg5).
- Next, I will transform the entire corpus to use VerbNet thematic role labels; and train an SRL system in that transformed space. I will evaluate this new SRL system in two ways:
  - By mapping the results of the new SRL system back to the space of PropBank labels, and comparing performance on PropBank Arg$n$ role labels.
  - By mapping the results of the baseline SRL system to the space of VerbNet labels, and comparing performance on VerbNet thematic role labels.

  The second evaluation is justified by the fact that VerbNet role labels may be more useful that PropBank labels. In particular, VerbNet thematic role labels

are more amenable to use for inferencing because they are not verb-specific.

- Finally, I will evaluate the SemLink mapping by comparing the effect of transforming via SemLink to the effect of transforming via alternative mappings, such as a mapping based on the role label descriptions in the PropBank frames files.

Chapter 6

Encoding Semantic Role Labelling Constraints

Chapter 5 showed that transforming the set of labels used to describe semantic roles could improve SRL performance by creating more coherent classes for local classifiers. However, SRL performance might also benefit from transformations that affect what type of *non-local* information is available to the classifiers. In particular, the use of output transformations can be examined with respect to allowing models to learn long-distance constraints and dependencies between arguments.

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 [17, 39, 52, 50, 54]. However, a single properly constructed joint model may outperform these re-ranking approaches.

6.1 Baseline System

My baseline system encodes the role labeling of a sentence for a given predicate as a sequence of word-aligned tags, describing the role played by each word in the

Word	He	saw	his	friend	John	break	the	window
Tag	Arg0	Pred	Arg1	Arg1	Arg1	Arg1	Arg1	Arg1
Tag	O	O	Arg0	Arg0	Arg0	Pred	Arg1	Arg1

Figure 6.1: **Sequence-Based SRL Encoding Example**. The first line of tags provides the annotation for the predicate "saw"; and the second line of tags provides the annotation for the predicate "break."

sentence. When using the PropBank label set, these tags are:

- `Arg0-Arg5`, `ArgM`: A word that is part of the specified argument.
- `Pred`: A word that is part of the predicate.
- `O`: Any other word.

When using theta role labels from the SemLink mapping, tags `Arg0-Arg5` would be replaced by tags `Agent`, `Patient`, etc. An example of this sequence-based encoding for semantic role labelings is given in Figure 6.1 The baseline system uses a linear-chain CRF to model tag sequences. I am still experimenting with the feature set for the baseline system.

6.1.1 Second Baseline System

I may also define a second baseline system, which walks over the sentence's parse tree rather than processing the sentence in linear left-to-right order. Thus, instead of assigning a tag to each word, we would assign a tag to each node in the tree. A node would be tagged as `Arg`n if it is the root of the parse tree node for argument `Arg`n, or any of its descendants. An example of this tree-based encoding is given in Figure 6.2.

This could be done one of two ways: either I could define a linear-chain CRF, where the input element list is a depth-first sequence of tree nodes; or I could define a tree-based model.

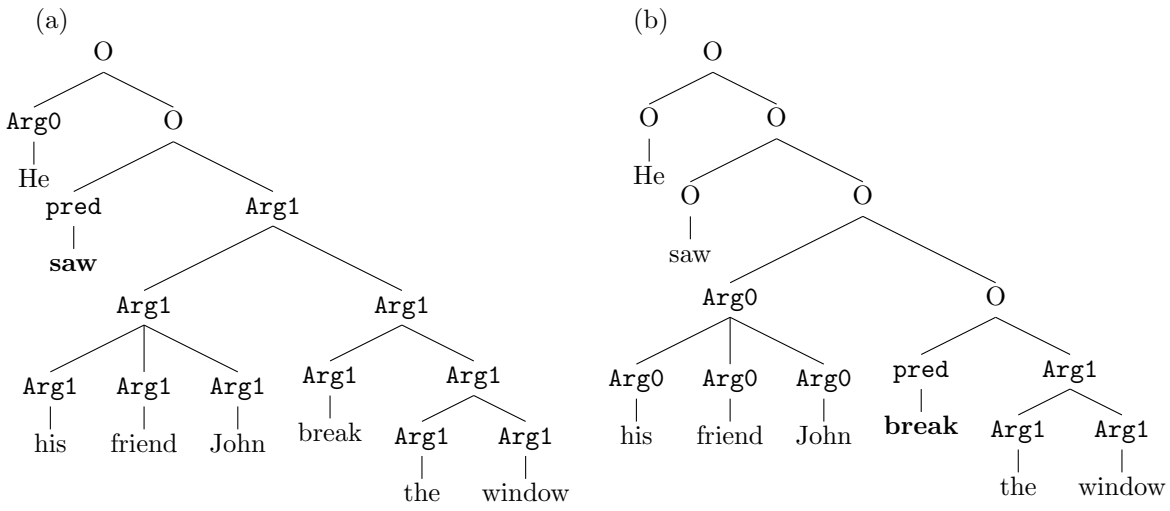


Figure 6.2: **Tree-Based SRL Encoding Example**. The left tree provides the annotation for the predicate "saw"; and the right tree provides the annotation for the predicate "break."

6.2 Encoding Long-Distance Dependencies

One deficiency of the baseline models described in the previous section is that they are incapable of learning dependencies between non-adjacent arguments. For example, it is typically unlikely for an agent (Arg0) argument to follow a theme (Arg1) argument if the predicate uses active voice. But the baseline systems have no way of learning this fact: by the time the system is examining a potential theme argument, the local model no longer has any record of whether an agent argument has been predicted or not.

However, by modifying the tag set, we can allow the system to keep track of a limited amount of history information, which can be used to learn long-distance dependencies. For example, if we augment each tag with information about whether or not an agent argument has been predicted, then the model will become capable of learning the dependency between agents and themes in active-voice sentences.

Although this transformation would allow the system to learn a new constraint, it is not obvious whether it would improve overall performance. Furthermore, there may be other constraints that are important to system performance, which might

require different transformations to the encoding. I therefore plan to apply the hill-climbing algorithm described in Chapter 4 to the baseline SRL systems. The FST modification operations already defined for the chunking task can be used as-is. It may also be helpful to add a few new modification operations that are specialized to the SRL task.

I will compare the performance of the system produced by the hill-climbing algorithm with both the performance of the baseline system, and the performance of existing systems that use re-ranking or other post-processing steps to enforce constraints.

Chapter 7

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 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, we can define a new combined classifier $\widehat{M}$ as follows:

$$\widehat{M}(y|x) = \sum_i w_i M_i(y|x) \tag{7.1}$$

We can then simply assign the highest scoring class to each new input value:

$$\widehat{y}^* = \arg\max_y \widehat{M}(y|x) \tag{7.2}$$

In this chapter, we will explore how weighted 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-learning tasks, but many of the results that we show for sequence-learning tasks could be generalized to other types of structured output tasks.

7.1 Local 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 sequences, 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 [48]. 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 7.1, 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 then the combined model would give the following sequence scores:

Model	**Sequence**	**P(Sequence)**
Combined Model	**P-N-V-N**	0.6
	P-V-N-N	0.2
	P-N-N-V	0.2

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

[1] (Assuming equal model weights.)

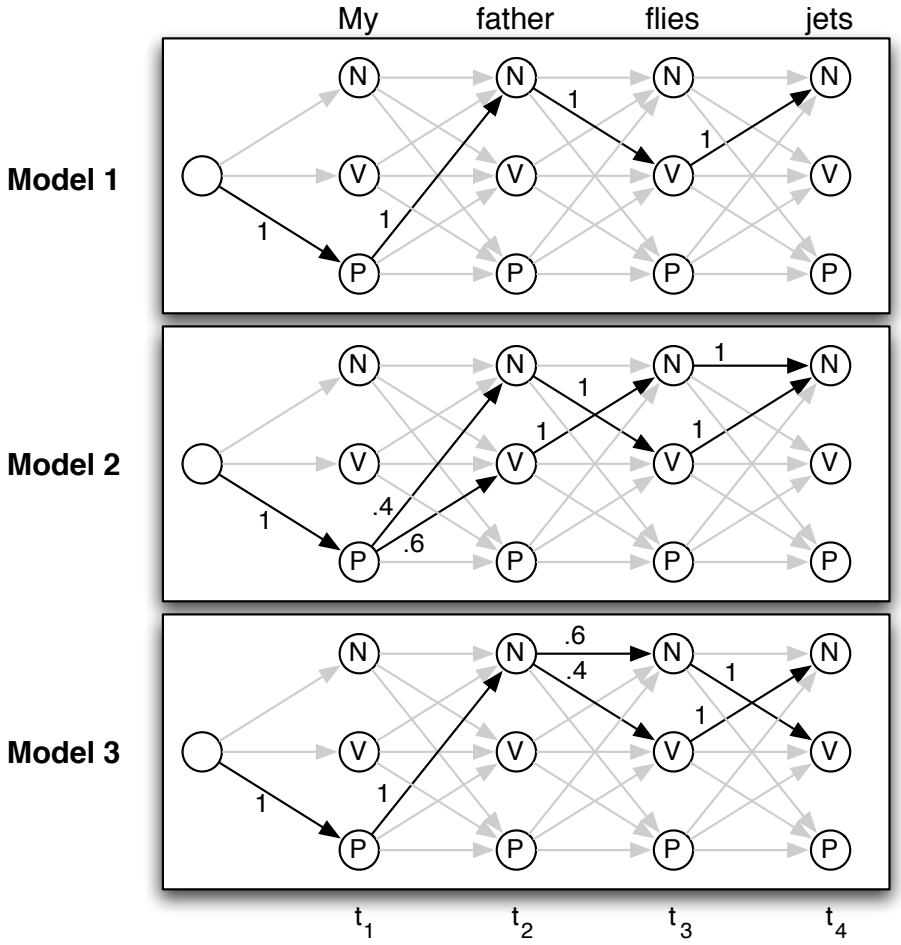


Figure 7.1: **Problematic Set of Models for Local Voting**. The Viterbi graphs generated by three different models for a given input sentence, *"My father flies jets."* (Note that words "father," "flies", and "jets" can each be used as either a verb or noun, depending on context.) The probability for 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 $1 \times 1 \times 0 \times 0 = 0$.

examples, there are generally an exponential number of sequences with nonzero probability, making it impractical to examine and rank them all.

7.2 Global Voting

What we would like, then, is to combine the models in such a way as to find the structured output with the best voted score. In particular, given a set of models M_i, where $M_i(\mathbf{y}|\mathbf{x})$ is the score assigned by M_i to structured output value $\mathbf{y}$ for input value $\mathbf{x}$, and a set of weights w_i for each model M_i, we can define a new combined classifier $\widehat{M}$ as follows:

$$\widehat{M}(\mathbf{y}|\mathbf{x}) = \sum_i w_i M_i(\mathbf{y}|\mathbf{x}) \tag{7.3}$$

In other words, we would like to use exactly the same equation that we use with classifiers. Unfortunately, finding the highest scoring output value is significantly more difficult when we are working with structured outputs, since there are a very large number of possible output values. Indeed, for many problem decomposition types, finding the highest scoring output value is intractable in the general case. However, it is possible to develop algorithms to find the best output in common cases; or to find an "approximate best" output in all cases.

In Section 7.3, we will develop a variant of the Viterbi Graph for expressing the global voting problem for Markovian sequence-learning tasks. 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. In the remaining sections, we will explore how voting can be performed using this framework.

7.2.1 Global Voting for Sequence-Learning Tasks

In sequence-learning tasks, the structured output value $\mathbf{y}$ is encoded as a sequence of tags. Given a set of models that share the same problem decomposition, we can

define a single unique encoding function that is shared by all models:

$$\begin{aligned} \text{encode}(\mathbf{y}) &= \vec{y} && (7.4) \\ &= (y_1, ..., y_T) && (7.5) \end{aligned}$$

Under these conditions, the global voting problem can be reformulated as the problem of finding the tag sequence that maximizes the weighted model score:

$$\begin{aligned} \mathbf{y}^* &= \text{decode}(\vec{y}^*) && (7.6) \\ \vec{y}^* &= \arg\max_{\vec{y}} \widehat{M}(\text{decode}(\vec{y}|\mathbf{x})) && (7.7) \\ &= \arg\max_{\vec{y}} \sum_i w_i M_i(\text{decode}(\vec{y})|\mathbf{x}) && (7.8) \end{aligned}$$

Recall that we are able to find the highest scoring output for a single model by constructing a Viterbi graph, and using dynamic programming to find the highest scoring path through that graph. Letting v^{M_i} be the Viterbi graph for model M_i given input x, Equation 7.8 can be rewritten as:

$$\vec{y}^* = \arg\max_{\vec{y}} \sum_i w_i v^{M_i}_{y_1}(1) \prod_{t=2}^{T} v^{M_i}_{y_{t-1}y_t}(t) \quad (7.9)$$

Unfortunately, Equation 7.9 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.

7.3 Grouped-State Viterbi Graphs

One interpretation of equation 7.9 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, $\{v^{M_i}\}$. This interpretation can be made concrete by combining the individual models' Viterbi graphs into a single graph, where the corresponding nodes

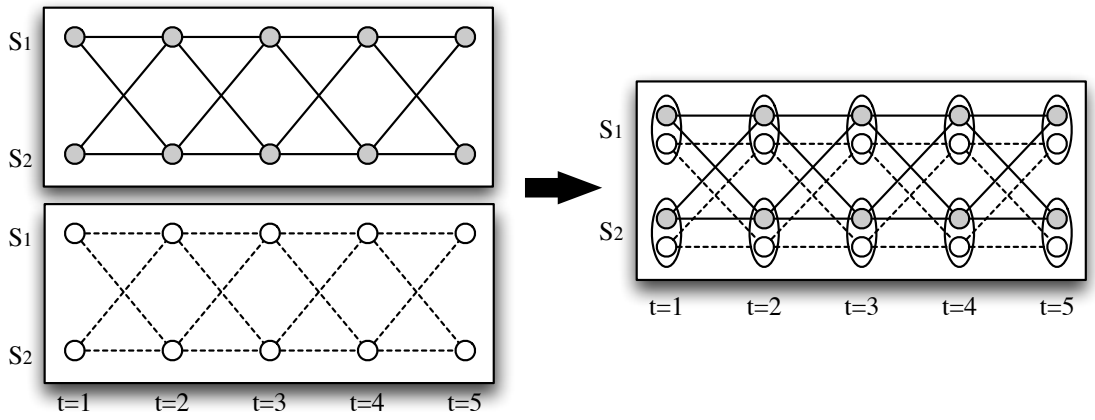


Figure 7.2: **Grouped-State Viterbi Graph**. The two graphs on the left are the Viterbi graphs generated by two models on a single input. We combine these two graphs into a new Grouped-State Viterbi Graph, on the right, where corresponding nodes from the original Viterbi graphs are grouped together.

from each graph are grouped together; and reformulating our goal as finding the highest-scoring sequence of *state groups*. An example of this new graph, which we will call a *Grouped-State Viterbi Graph*, is shown in Figure 7.2.

A Grouped-State Viterbi Graph is very similar to a simple Viterbi graph, except that we are interested in answering questions about sequences of state groups, instead of sequences of simple states. In particular, we will be interested in determining the overall score for a given sequence of state groups; and in finding the state group sequence that maximizes the overall score.

Formally, we define a Grouped-State Viterbi Graph to consist of a Viterbi graph $\langle S, T, Q, E \rangle$, augmented by a set of state groups $G = g_1, ..., g_K$, such that each state $s \in S$ is contained in exactly one group $g \in G$. We define the score of a state group sequence $\vec{r} = (r_1, ..., r_T)$ to be the sum of the scores of all state sequences $\vec{y}$ that are

Grouped-State Viterbi Graph	$\langle S, T, Q, E, G \rangle$		
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	$=$	$\{g_1, ..., g_K\}$
State Sequence	$\vec{y}$	$=$	$(y_1, ..., y_T)$
State Group Sequence	$\vec{r}$	$=$	$(r_1, ..., r_T)$

Figure 7.3: **Notation for Grouped-State Viterbi Graphs**.

consistent with it:

$$\text{score}(\vec{r}) = \sum_{\vec{y}:\forall t, y_t \in r_t} \text{score}(\vec{y}) \tag{7.10}$$

$$= \sum_{\vec{y}:\forall t, y_t \in r_t} \left(\text{weight}(q_0 \rightarrow q_{1,y_1}) \prod_{t=2}^{T} \text{weight}(q_{t-1,y_{t-1}} \rightarrow q_{t,y_t}) \right) \tag{7.11}$$

7.3.1 Sequence Voting with a Grouped-State Viterbi Graph

In order to use Grouped-State Viterbi Graphs to perform global voting for a sequence-learning 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:

$$G = \{g_s : s \in S\} \tag{7.12}$$

$$g_s = \{\langle M_i, s \rangle : M_i \in models\} \tag{7.13}$$

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 weight(M_i):

$$\text{weight}\left(q_0 \rightarrow q_{1,\langle M_i,s\rangle}\right) = \left(\text{weight}(M_i)\right)\left(v_s^{(M_i)}(1)\right) \quad (7.14)$$

$$\text{weight}\left(q_{t-1,\langle M_i,s\rangle} \rightarrow q_{t,\langle M_i,s'\rangle}\right) = v_{ss'}^{(M_i)}(t) \quad (7.15)$$

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 7.10, 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:

$$\text{score}(\vec{y}^*) = \max_{\vec{y}} \widehat{M}(\text{decode}(\vec{y})|\mathbf{x}) \quad (7.16)$$

$$= \max_{\vec{y}} \sum_i w_i \text{score}^{(M_i)}(\vec{y}) \quad (7.17)$$

$$= \max_{\vec{r}} \left(\text{score}(\vec{r})\right) \quad (7.18)$$

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.

7.4 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

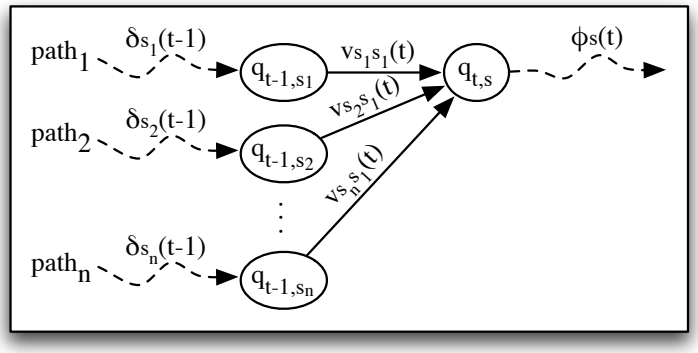


Figure 7.4: **Viterbi Dynamic Programming Decomposition.** A portion of a Viterbi graph, showing the score decomposition that is used by the Viterbi algorithm to calculate $\delta_s(t)$ recursively.

how this algorithm can be made to cover all inputs, at the expense of producing a near-optimal path.

7.4.1 Why it's Hard

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

Figure 7.4 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

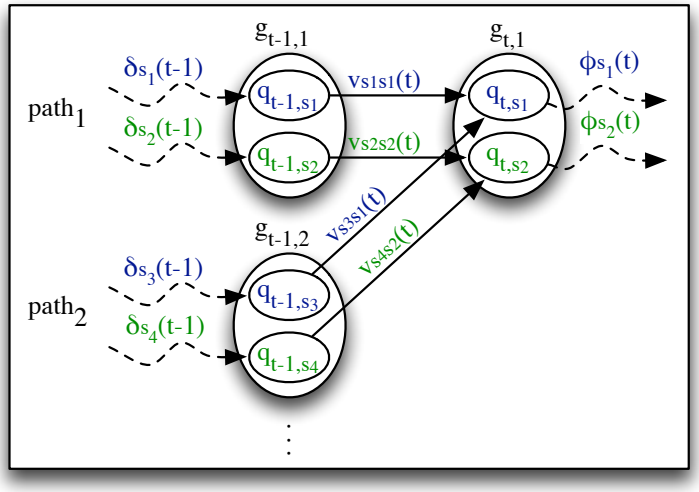


Figure 7.5: **Choosing the Best Path in a Grouped State Viterbi Graph**. This diagram shows a portion of a Grouped State Viterbi Graph, illustrating the difficulty in choosing the best incoming path. This diagram corresponds to the portion of a Viterbi Graph shown in figure 7.4.

source nodes is part of the highest scoring path, note that:

$$\begin{aligned} \arg\max_{path_i}(\text{score}(path_i)) &= \arg\max_{path_i} \delta_{s_i}(t-1)v_{s_i s}(t)\phi_s(t) && (7.19) \\ &= \arg\max_{path_i} \delta_{s_i}(t-1)v_{s_i s}(t) && (7.20) \end{aligned}$$

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 7.4 to Figure 7.5, 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 which of these group-node paths has the highest score:

$$\arg\max_{path_i}(\text{score}(path_i)) = \arg\max_{path_i} \sum_{s' \in g_{t-1,i}} \sum_{s \in g_{t,1}} \delta'_s(t-1)v_{s's}(t)\phi_s(t) \quad (7.21)$$

But there is crucial difference between Equation 7.19 and Equation 7.21: $\phi_s(t)$ 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 7.5. 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.

7.4.2 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:

$$R_g(t) = \phi_{s_1}(t)/\phi_{s_2}(t) \qquad (g = \{s_1, s_2\}) \tag{7.22}$$

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

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

7.4.3 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 7.6 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 7.6 (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 7.6 (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

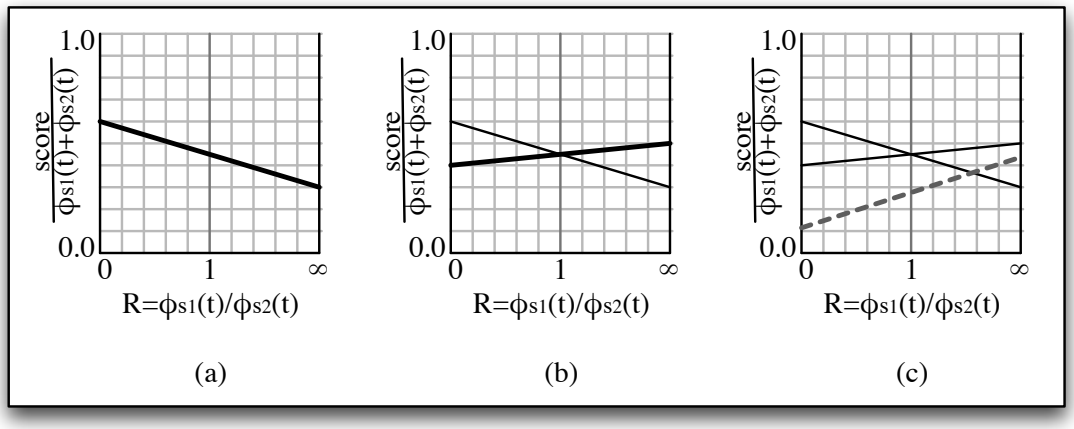


Figure 7.6: **Pruning Candidate Incoming Paths with** R. In graph (a), we plot R vs. the overall score for an incoming path, given that value of R. In graphs (b) and (c), we plot the same function for two more incoming paths. Since the line segment added in graph (c) is not maximal for any value of R, 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.

added in (b) iff $R < 0.95$ (the crossover point).[2] In Figure 7.6 (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.

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

Nodes per Grouped State	Avg # Elements	Max # Elements
1	1	1
2	8.3	24
3	48.2	123
4	89.3	609

Table 7.1: **Number of Line Segments or Hyperplanes in the Pruning Graph**.

Experimental Results

In order to test the hypothesis that the number of line segments stays manageable, I constructed a series of randomly generated Grouped State Viterbi graphs, and found the highest scoring group sequence in each graph, using a variant of the Viterbi algorithm that uses the pruning techniques described in this section. Table 7.1 lists the results.

7.4.4 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 7.7 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 7.7 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

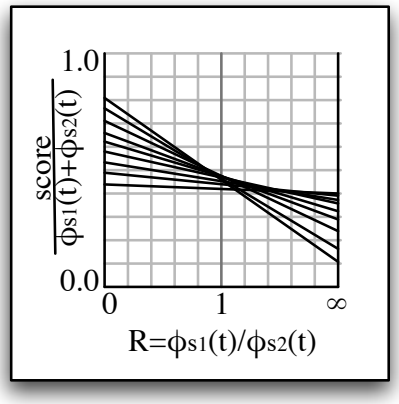


Figure 7.7: **Problematic Case for Pruning**.

pruning will lower the score of the value we find, compared to the optimal value. To understand how, see Figure 7.8, where there are three incoming paths contributing to the pruning graph. Consider the case where we prune path b. In the worst case, 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.

7.5 Global Voting with Multiple Encodings

Sections 7.3 and 7.4 showed how multiple models that all use the same encoding could be combined into a single model via global voting. In this Section, I will show how this approach can be extended to the case of combining multiple models with

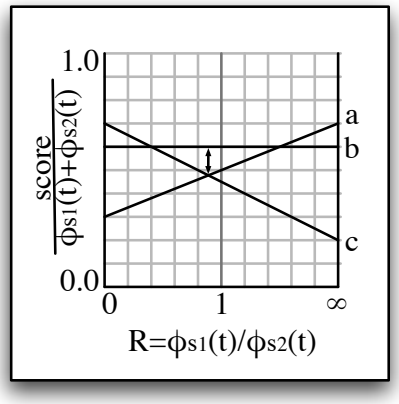


Figure 7.8: **Using Pruning to find the Approximate-Best Value**.

different encodings. 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. The basic approach that we will use to combine the individual models is:

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 state groups correspond to states in the canonical representation.

3. Combine these Grouped State Viterbi Graphs into a single graph, by combining the individual graph's state groups and copying the transitions from the individual graphs.

4. Apply the pruning algorithm described in Section 7.4 to find the optimal group sequence in the combined Grouped State Viterbi Graph.

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.[3] The FST

[3]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.

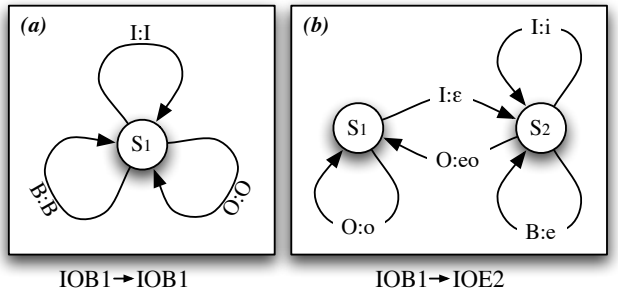


Figure 7.9: **FSTs Representing the `IOB1` and `IOE2` Encodings**. We will use these two encodings as an example to illustrate the global voting algorithm with multiple encodings. Both FSTs are expressed with respect to the canonical encoding `IOB1`.

representations for these two encodings are shown in Figure 7.9. 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 7.10.

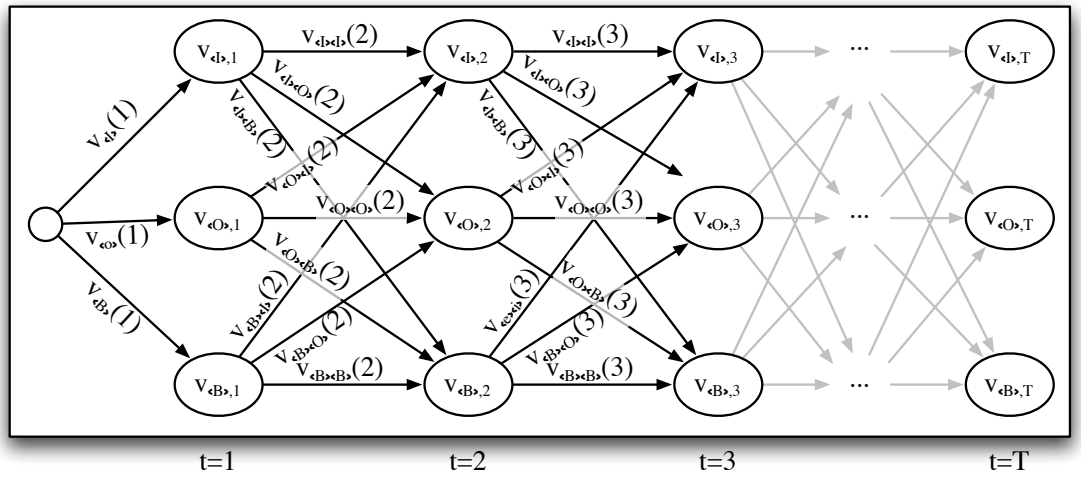
7.5.1 Transforming Viterbi Graphs into Canonical Grouped State Viterbi Graphs

Before we can perform global voting between models that use different encodings, we must first convert their Viterbi graphs into a common format. This will allow us to combine their individual Viterbi graphs to form a new graph where each model's probability estimate for a given value is represented by the same path (through group nodes).

I will refer to this common format as the *Canonicalized Viterbi Graph* for a given model. It consists of a Grouped State Viterbi Graph, with the following three properties:

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

IOB



IOE

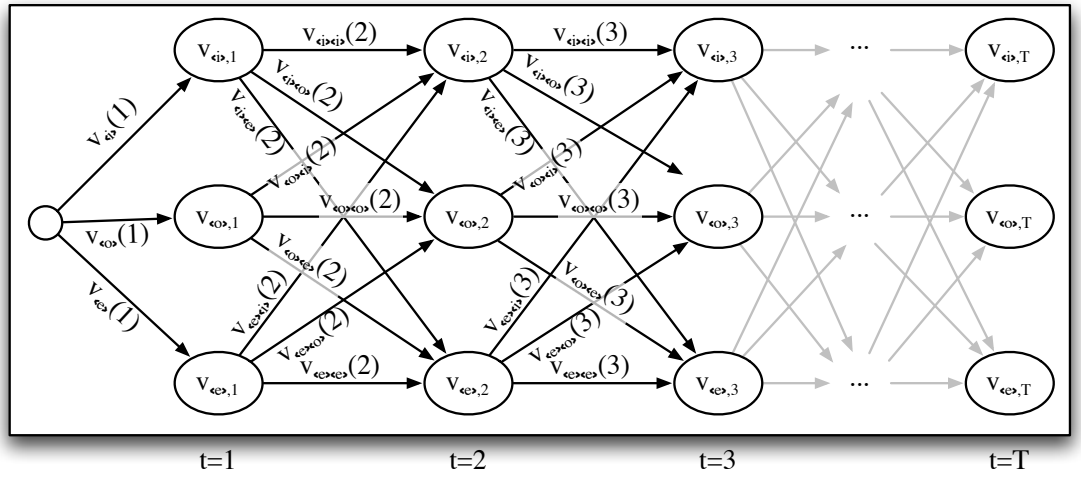


Figure 7.10: **Viterbi Graphs for the `IOB1` and `IOE2` Encodings**. These two Viterbi graphs can not be directly combined, because they use incompatible states: the `IOB1` graph uses the three states `I`, `O`, and `B`, while the `IOE2` graph uses the states `i`, `o`, and `e`. We must therefore transform the graphs into a common format before they can merged.

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

3. For each path p, the corresponding value $value(p)$ is encoded by the sequence of group node labels that the path passes through.

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 Canonicalized Viterbi Graph, 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. The two FSTs in 7.9 are already normalized.

We will represent each output value in the Canonicalized Viterbi Graph 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 Canonicalized Viterbi Graph 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 Canonicalized Viterbi Graph 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 Canonicalized Viterbi Graph.
- The t index of the most recently consumed canonical tag. This determines in which time slice the node will be located in the new Canonicalized Viterbi Graph.

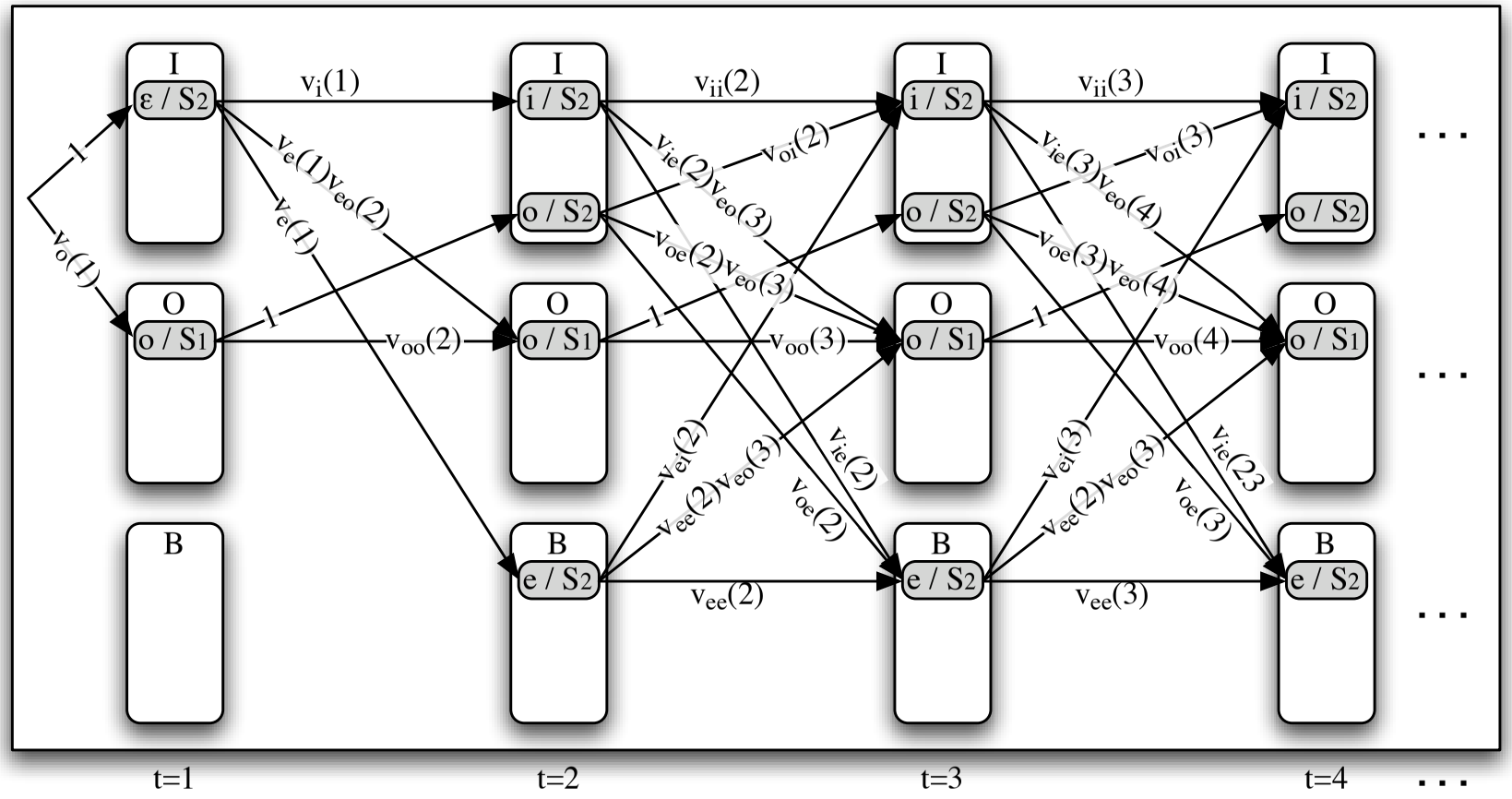


Figure 7.11: **Canonicalized Viterbi Graph for an `IOE2` model**. The Canonicalized Graph contains one grouped state for each tag in the canonical encoding, allowing it to be combined with other models' Canonicalized Graphs. Each node in the graph is a tuple $\langle s, t, tag_c, tag_e \rangle$, where 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 Canonicalized Graph 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 7.12; and the resulting Canonicalized Viterbi Graph for an `IOE2` model is shown in Figure 7.11.

7.5.2 Combining Canonical Grouped State Viterbi Graphs

Once each model's Viterbi Graph has been converted to a Canonical Grouped State Viterbi Graph, we can perform weighted voting by simply merging the individual Canonical Graphs into a single graph. In particular, the merged graph's nodes and edges are the union of the individual models' Canonical Graphs; and the groups are formed by merging the corresponding groups from each of the models. If we wish to apply different weights to the different models, we can do so by modifying each graph's initial edge weights to include the model weightings, as we did in the case of global voting between models with a single encoding.

Once the merged graph has been created, we can apply the algorithms described in Section 7.4 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 weighted-voting model.

7.6 Proposed Work

For my dissertation, I plan to apply the global voting methods described in this Chapter to the chunking system described in Chapter 4 and to the SRL system

```
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 7.12: **Canonical Grouped State Viterbi Graph Construction Algorithm**.

described in Chapter 6. I will then compare the performance of these voting models to the individual underlying models; and to models that perform voting locally rather than globally. I also plan to try adapting the hill-climbing algorithm described in Chapter 4 to produce multiple complementary encodings, by searching for a set of encodings that individually perform well, but that are substantially different from one another.

Chapter 8

Adding Non-Local Output Features to Structured Viterbi Models

8.1 Introduction

The effects of output encoding transformations can be divided into two general classes: local effects, which influence the difficulty of learning individual sub-problems, and global effects, which determine the model's ability to learn long-distance dependencies. This chapter explains how the output encoding affects which long-distance dependencies can be learned; and shows how transformations to the output encoding can expand the set of long-distance dependencies a model can learn (a positive global effect). When done in a naive way, these transformations would split the data for individual sub-problems, aggravating the sparse data problem (a negative local effect). But by being careful about exactly how the transformation is done, it is possible to achieve the positive global effects while avoiding the negative local effects.

8.2 Structured Viterbi Models

Hidden Markov Models, PCFGs, Conditional Random Fields, and MEMMs are all members of a class of probabilistic models known as *Structured Viterbi Models*[1]. These models define a unique graphical structure corresponding to each possible output value. They specify the probability of a given output by first computing local scores that evaluate the plausibility of individual nodes or cliques in the graph; and then combining those scores, generally via multiplication or addition.

One disadvantage shared by all models in this class is that they may not contain features which depend on non-local pieces of output structure. More precisely, the score of an output may not depend in any non-compositional way on any two nodes that are not connected in the graph representing that output. Of course, we can get around this restriction by adding edges to the graph; but doing so can result in intractable learning and prediction problems. And in order to express some non-local features, we would essentially need to fully connect the graph. Examples of features that depend on the output in non-local ways include the following. (Notation is summarized in Figure 8.1.)

In a sequence tagging task:

- Are there any elements with tag T_i to the left of this tag?
- Are there any elements with tag T_i to the right of this tag?
- How many elements with tag T_i are to the left of this tag?
- How many elements intervene between this element and the closest element that was tagged with T_i?
- What was the lexical item of closest element to the left that was tagged T_i?

In a parsing task:

- Is this constituent dominated (at any distance) by a node with label T_i?

[1]The word *Viterbi* is used here to distinguish this class of models from other types of Structured prediction models, such as transformation-based models (e.g., Brill's tagger).

The set of node labels	T	=	$\{T_i\}$
Augmented node labels	T^+	=	$\{T_i[p_1{=}v_{i,1}, \ldots, p_k{=}v_{i,k}]\}$
Input value (sequence tasks)	$\vec{x}$	=	$(x_1, \ldots, x_T)$
Output value (sequence tasks)	$\vec{y}$	=	$(y_1, \ldots, y_T)$ $\quad (y_t \in T \text{ or } y_t \in T^+)$

Figure 8.1: **Notation for Augmenting Node Labels**.

- GPSG-style "slashed category" features.

This chapter presents a method for adding features that depend on non-local pieces of output structure to Structured Viterbi Models.

8.3 Transforming the Output Graph

An important common feature of Structured Viterbi Models is that all scores are computed over local graphical regions (nodes or cliques), and then combined using some simple technique (multiplication or addition). Thus, the only way that we can add features that depend on non-local aspects of the output will be to make those features available at local regions. This can be done by transforming the graphical representation we use for output values.

8.3.1 Adding Edges

As was mentioned in the introduction, the simplest way to turn non-local features into local ones is by simply adding edges to the graph. However, this approach comes at a cost: the more densely-connected we make the graph structures, the more likely it becomes that training and prediction will be intractable. For example, Figure 8.2 shows that adding a relatively simple feature can easily result in a fully-connected graph.

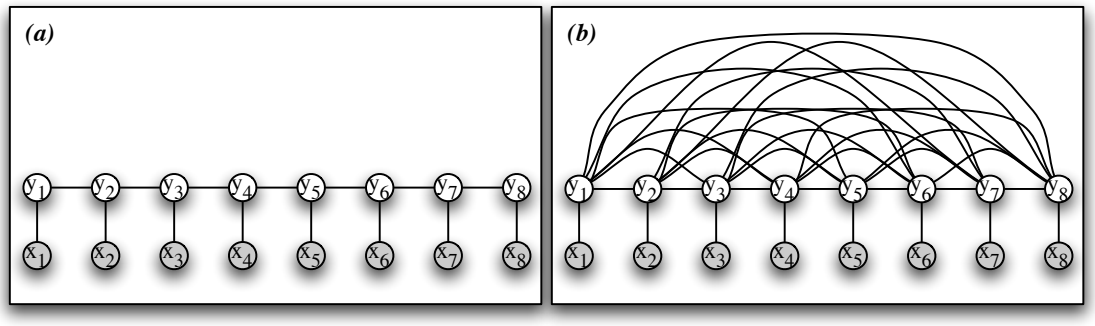


Figure 8.2: **Adding Edges**. The graph in (a) shows a typical output graph for a sequence tagging model. The graph in (b) shows what edges we would need to add if we wanted to include a feature that answers the question "are there any elements with tag T_i to the left of this tag" at each node. In particular, this question can not be answered without looking at the tags of all output elements that have been generated before; so we must add an edge from each element to every element that precedes it, resulting in a fully-connected graph.

8.3.2 Augmenting Node Labels

But adding edges isn't our only option. In fact, we can transform the output value graph in *any* way we wish, as long as each output value is encoded by exactly one graph. Thus, one approach to the problem of adding non-local output features is to add information content to the labels in a graph. Information can then be propagated around the graph in a global manner, using chains of local regions as a conduit. In particular, we can subdivide each existing label T_i into new labels $T_i[p{=}v_1], \ldots, T_i[p{=}v_n]$, where p is some augmentation variable that we wish to propagate; and $v_1, \ldots, v_n$ is the set of values that p may take.

For example, if we want to add a feature that answers the question "are there any elements with tag T_i to the left of this tag," then we can do so by subdividing each existing tag T_j into new tags $T_j[seen(T_i){=}True]$ and $T_j[seen(T_i){=}False]$. After modifying the output graphs, the information needed for the desired feature is available locally; and so the feature can be added. In particular, the feature should be true for tags of the form $T_j[seen(T_i){=}True]$; and false for tags of the form

$T_j[seen(T_i)=False]$.

8.4 Training with Augmented Node Labels

Although augmenting node labels allows us to add non-local output features, it also has implications for training: if we are not careful, then splitting node labels will split our training data, leading to an increase in sparse data problems.

The simplest way to train a model with augmented node labels is to treat each augmented label as an atomic constant; and use the existing learning methods. For example, if we wish to augment our model with the variable $seen(T_i)$, described above, then we would replace every tag T_j in the training data with $T_j[seen(T_i)=True]$ if there is at least one T_i preceding it; and with $T_j[seen(T_i)=False]$ otherwise. We would then apply the original learning method to the transformed training corpus. Using this simple approach, the learning method has no knowledge of the internal structure of the augmented labels. This gives rise to two problems:

1. Constraints on the labels of adjacent nodes are not captured.

2. Feature weight estimates are based on augmented labels, which may not have enough training data to support accurate learning (especially for augmentations involving lexical information).

8.4.1 Constraints

There are typically strong constraints on the augmented labels of adjacent nodes. For example, the *seen* variable will always follow the local constraints listed in Figure 8.3. Any output value that does not meet these constraints should be assigned a probability of zero. However, these constraints are not directly available to the learning model; it will need to learn them from scratch. This increases the number of parameters the model must learn, which correspondingly increases the amount of

1. For the first element, $seen(T_i)$ is false. I.e.:
   $y_1 = T_j[seen(T_i)=False]$
2. If the previous tag was T_i, then $seen(T_i)$ is true. I.e.:
   if $y_t = T_i[seen(T_i)=v]$ **then** $y_{t+1} = T_j[seen(T_i)=True]$
3. If the previous tag was not T_i, then $seen(T_i)$ is copied from the previous augmented tag. I.e.:
   if $y_t = T_j[seen(T_i)=v]$ **and** $T_j \neq T_i$ **then** $y_{t+1} = T_k[seen(T_i)=v]$

Figure 8.3: **Constraints on** *seen*. Any output value that does not meet these constraints should be assigned a probability of zero. In this example, the constraints completely determine the value of the augmentation variable; however, this need not be the case in general. Also, note that the constraints do not have any inherent notion of "direction" – this is necessary if they are to be used with undirected models like CRFs.

training data needed to accurately model the task. And in cases where augmentation variables can take on a large number of values (e.g., lexical information), the learning model may never fully learn the constraints.

An analogous problem arises when we attempt to use the model for prediction. Since the dynamic programming algorithms used for prediction do not have access to the constraints, they must explore all possible sequences of augmented tags. But the total number of augmented tags can grow quite rapidly: if each tag is augmented with k variables, each taking n values, and the graph contains cliques of m nodes, then the running time for prediction will be increased by a factor of $(n^k)^m$.

The solution is fairly straight-forward: we should provide the learning model and the prediction algorithm with information about the constraints on each augmentation variable's values.[2] In the case of the learning model, this can be done by fixing the weights between any two adjacent non-compatible tags to $-\infty$. In the case of the prediction algorithm, this can be done by exploring only those tag sequences that are consistent with the constraints.

[2]This has the additional benefit that we don't need to worry about what to do if an "invalid output" is generated by the learned model.

8.4.2 Data Splitting

The second problem with treating the augmented tags as atomic constants is that it results in a significant increase in data sparsity. In particular, splitting a tag implies splitting the training data for that tag. If each tag is augmented with k variables, each taking n values, then splitting the tags will decrease the amount of training data available for each tag by a factor of n^k (on average). These splits will tend to decrease the accuracy of the learned model because although feature weights depend to some extent on a node's annotations, they usually depend much more strongly on the node's base label. But with appropriate adjustments to the learning model's local scoring function, it is not necessary to actually split the training data.

Avoiding Sparse Data in HMMs

Recall that all Structured Viterbi Models use local scoring functions to compute the plausibility of individual cliques in the graph; and then combine those scores. HMMs define the overall score of an output to be the product of local scores, where local scores have the forms $P(y_t|y_{t-1})$ (for adjacent output nodes) and $P(x_t|y_t)$ (for adjacent input and output nodes):

$$P(x, y) = \prod_t P(y_t|y_{t-1})P(x_t|y_t) \tag{8.1}$$

When we augment the node labels (y_i), we are essentially increasing the size of the distributions $P(y_t|y_{t-1})$ and $P(x_t|y_t)$. In particular, the range of y_i is expanded from T to T^+, where $|T^+| \approx n^k|T|$. If we continue to use the same local models for these distributions[3], then the amount of data available to train each condition will decrease. But we can allow the local models to share data between augmented labels

[3]Typically MLE for $P(y_t|y_{t-1})$ and Naive Bayes for $P(x_t|y_t)$.

simply collapsing out the distinctions made by the augmentations:

$$P(y_t|y_{t-1}) \approx P(strip(y_t)|strip(y_{t-1})) \quad (8.2)$$

$$P(x_t|y_t) \approx P(x_t|strip(y_t)) \quad (8.3)$$

$$strip(T_i[\cdots]) = T_i \quad (8.4)$$

In a sense, this appears to be a step backward. It brings us back to where we started from: the local models will be modelling exactly the same set of data that they did in the original (unaugmented) system. Collapsing out the augmentations makes them unavailable to the local models, so we can't use them to define features.

But now we have access to two local models: an augmented model, which can use the non-local features but may suffer from data sparsity; and a collapsed model, which does not have access to the new features but which is more robust. We can combine these two models via local voting (e.g., linear interpolated back-off).

If we add multiple augmentation variables to the output structures, then we can collapse out different sets of distinctions, and apply voting to the resulting models. E.g., we could include a model for each augmentation variable p_j, which approximates the local distributions as:

$$P(y_t|y_{t-1}) \approx P(strip_{p_j}(y_t)|strip_{p_j}(y_{t-1})) \quad (8.5)$$

$$P(x_t|y_t) \approx P(x_t|strip_{p_j}(y_t)) \quad (8.6)$$

$$strip_{p_j}(T_i[\ldots, p_j = v_{i,j}, \ldots]) = T[p_j = v_{i,j}] \quad (8.7)$$

It's worth noting that a number of augmented PCFG parsing models (such as Collins' parser) can be described in exactly these terms: they augment node labels with a variety of non-local information, such as the head word, the head word's part of speech, and subcategorization frame information; they define constraints between those pieces of information; and they use some form of local voting to exploit the trade-off between the lower bias of more fine-grained local models with the lower variance of more general local models.

Avoiding Sparse Data in CRFs

We can use a similar approach to avoid the sparse data problems that arise from splitting tags in Conditional Random Fields. CRFs define the overall score of an output to be (the log of) the sum of local scores, where each local score is the sum of a weighted feature vector:

$$P(y|x) \propto \log\left(\sum_{c \in cliques} \sum_{i} w_i f_i(c)\right) \tag{8.8}$$

Feature weights are set by maximizing the total log likelihood of the training data. Typically, features are defined as binary functions pairing a property of the local input value with a set of node labels (one for each node in the clique). For example, in a typical linear chain CRF, each feature has the form:

$$f(\langle x_t, y_{t-1}, y_t\rangle) = \begin{cases} 1 & \text{if } \mathit{property}(x_t) \text{ and } \langle y_{t-1}, y_t\rangle = \langle T_i, T_j\rangle \\ 0 & \text{otherwise} \end{cases} \tag{8.9}$$

Where *property* is some property of the input element x_t, and T_i and T_j range over all tags. Since a feature is defined for every pair of node labels, augmenting the node labels will increase the number of features. In particular, the number of features per property will increase from $|T|$ to $|T^+|$, where $|T^+| \approx n^k|T|$. Each of these new features will be supported by correspondingly less training data.

But CRFs are not required to use features of the form shown in equation (8.9). In principle, CRF features $f_i(c)$ may be defined as *any* function over the clique c. Thus, we can collapse out the distinctions made by augmentations, just as we did with HMMs, by defining features using $\mathit{strip}(T_i)$:

$$f(\langle x_t, y_{t-1}, y_t\rangle) = \begin{cases} 1 & \text{if } \mathit{property}(x_t) \text{ and } \langle y_{t-1}, y_t\rangle = \langle \mathit{strip}(T_i), \mathit{strip}(T_j)\rangle \\ 0 & \text{otherwise} \end{cases} \tag{8.10}$$

As was the case with HMMs, this returns us to our original (unaugmented) model. I.e., defining an augmented model whose features are conditioned on stripped tags

is identical to using the unaugmented model. But unlike HMMs, CRFs can directly combine the augmented features with the collapsed features in a single model. Thus, there is no need to define separate local models, and combine them using voting; instead, we can simply provide the CRF with some features that depend on augmented labels, and other features that depend on collapsed labels.

If we add multiple augmentation variables to the output structure, then we can define features that collapse out different sets of distinctions. E.g., we could include features with the following forms:

$$f(\langle x_t, y_{t-1}, y_t \rangle) = \begin{cases} 1 & \text{if } \mathit{property}(x_t) \text{ and} \\ & \quad \langle y_{t-1}, y_t \rangle = \langle \mathit{strip}_{p_j}(T_i), \mathit{strip}_{p_j}(T_j) \rangle \\ 0 & \text{otherwise} \end{cases} \tag{8.11}$$

$$f(\langle x_t, y_{t-1}, y_t \rangle) = \begin{cases} 1 & \text{if } \mathit{property}(x_t) \text{ and} \\ & \quad \langle y_{t-1}, y_t \rangle = \langle \mathit{strip}(T_i), \mathit{strip}(T_j) \rangle \\ 0 & \text{otherwise} \end{cases} \tag{8.12}$$

$$\mathit{strip}_{p_j}(T_i[\ldots, p_j = v_{i,j}, \ldots]) = T[p_j = v_{i,j}] \tag{8.13}$$

$$\mathit{strip}(T_i[\cdots]) = T_i \tag{8.14}$$

8.5 The Cost of Augmenting Node Labels

The techniques described so far allow us to add features that depend on output in a non-local way without un-necessarily increasing the number of parameters that the model needs to learn. But although these techniques will not decrease the performance of the model (like more naive methods would), they *will* increase the running time needed for both training and prediction.

In particular, the dynamic programming algorithms used for prediction will need to be run over augmented tags. Since the running time of these algorithms is proportional to $|T|^m$ (where m is the clique size), augmenting the tags could in principle

increase the running time for prediction by as much as $(n^k)^m$. However, the constraints associated with many augmentation variables will uniquely define (or at least severely restrict) the possible values for one node label's annotations, given the others. Thus, the increase in running time is more likely to be on the order of $n^{k(m-1)}$. This increase in running time also affects iterative learning models like CRF, that must perform tasks similar to prediction at each iteration.

Thus, we must be careful to limit the number of augmentation variables we add to the output structure. But even a handful of augmentation variables may be sufficient to allow the model to capture critical long-distance dependencies. And given a small set of augmentation variables, we can define a large variety of features that depend on those augmentations in different ways.

8.6 Summary of the Proposed Algorithm

8.6.1 Training

We can train a model that uses features with non-local dependencies on the output value using the following procedure:

1. Define one or more augmentation variables that provide the desired information at the appropriate nodes.
    - Given an output value, the value of each augmentation variable must be uniquely defined for each node.
    - It must be possible to uniquely define the correct augmentation variable values for all nodes using only local constraints. A *local constraint* is a binary function of the augmented node labels and input values of a local region (clique) of the graph.
2. Transform the training corpus to use the augmented variables.

3. Add constraints to the learning models by fixing the weights between any two adjacent non-compatible labels to $-\infty$.

4. Modify the learning models' local models to include scoring functions over both the augmented node labels and the collapsed node labels.

    - For HMMs and PCFGs: define a set of local models that estimate probabilities based on counts of different combinations of augmented and collapsed node labels; and combine them with voting methods.
    - For CRFs: define features based on different combinations of augmented and collapsed node labels.

5. Train the modified model using the transformed training corpus.

8.6.2 Prediction

We can then use that model to predict the most likely output value for new inputs by using the following procedure:

1. Use a dynamic programming algorithm to run the trained model on a new input value.

    - Use the constraints to limit which node labels the dynamic programming algorithm explores at each step.

2. Discard the annotations, and return the output value corresponding to the resulting graph.

8.7 Conclusions

In Structured Viterbi Models, output node labels serve (at least) two purposes: first, they are used as a parameter to define local scoring functions; and second, they

are used as an information channel, to allow information to flow between different locations in the graph. Furthermore these two purposes are separable. In particular, it is possible to enrich the information channels without splitting the parameters of the local scoring functions. Of course, the parameters of local scoring functions will need to be expanded to some extent, to allow them to use the enriched information channels; but this can be done in a controlled way, rather than by simply splitting all parameters.

8.8 Proposed Work

For my dissertation, I plan to evaluate the effect of this algorithm, on both performance and run-time, by using augmentation variables to add manually selected classes of non-local features to existing state-of-the-art linear chain CRF systems. I also plan to combine this algorithm with the hill-climbing system that was described in Chapter 4. In particular, constraints can be defined based on the FST that transforms canonical encodings to new encodings; and operations that modify an FST by replacing an existing label with a new label, such as output relabeling and feature specialization, can be changed to augment the label rather than replace it. This should improve the hill-climbing system's performance, by allowing it to introduce new long-distance output dependencies without needing to split the training data for local sub-problems.

Chapter 9

Summary of Proposed Work

9.1 Improving Chunk Encodings via Hill Climbing

As discussed in Chapter 4, the performance of state-of-the-art NP chunkers is very close to the upper limit imposed by inter-annotator accuracy. I therefore plan to focus on the task of Biomedical Named Entity Detection, where there is more headroom for improvement. I will adapt my hill-climbing system to this new domain by defining a new CRF chunker, with features based on state-of-the-art Biomedical Named Entity systems; and modifying the hill-climbing system to use this new chunker.

One issue with the current hill-climbing algorithm is that its search is fairly undirected: it simply chooses modification operations at random. I therefore plan to carry out a number of experiments designed to provide the hill-climbing search algorithm with more direction. First, I will perform several experiments that explore factors influencing the impact of existing modification operations on system performance; and use those experiments to provide improved guidance to the hill-climbing search. I will then experiment with the use of problem-driven learning and unsupervised clustering to provide additional guidance to the hill-climbing direction.

9.2 Transforming Semantic Roles via SemLink

Chapter 5 described several preliminary experiments which suggest that mapping PropBank semantic roles to a more coherent set of role labels may improve both performance and the ability to generalize for SRL systems. For my dissertation, I will further explore this hypothesis by performing several additional experiments that evaluate the effect of transforming role labels via the SemLink mapping. First, I will repeat the experiments described in Section 5.5.2 on the remaining arguments (Arg0 and Arg3-Arg5). Next, I will transform the entire corpus to use VerbNet thematic role labels; and train an SRL system in that transformed space. I will evaluate this new SRL system in two ways: first, I will try mapping its results back to the space of PropBank labels, and compare it with the baseline system with regard to PropBank labelings; and second, I will apply the mapping to the output of the baseline SRL system, and compare the performance of the two systems in the transformed space of VerbNet thematic role labelings. This second evaluation is justified by the fact that VerbNet role labels may be more useful that PropBank labels: since they are not verb-specific, they are more amenable to use with inferencing. Finally, I will evaluate the SemLink mapping by comparing the effect of transforming via SemLink to the effect of transforming via alternative mappings, such as a mapping based on the role label descriptions in the PropBank frames files.

9.3 Encoding Semantic Role Labeling Constraints

Chapter 6 described two baseline SRL systems which are incapable of directly learning constraints and dependencies between different arguments. However, by transforming the output encoding used by these systems, we can provide them with the information necessary to learn these long-distance relationships. I therefore plan to apply the hill-climbing algorithm described in Chapter 4 to these two baseline SRL systems. The FST modification operations already defined for the chunking task

will be used as-is. It may also be helpful to add a few new modification operations that are specialized to the SRL task. I will then compare the performance of the system produced by the hill-climbing algorithm with the performance of the baseline system, and with the performance of existing systems that use re-ranking or other post-processing steps to enforce constraints.

9.4 Global Voting

For my dissertation, I plan to apply the global voting methods described in Chapter 7 to the chunking system described in Chapter 4 and to the SRL system described in Chapter 6. I will then compare the performance of these voting models to the individual underlying models; and to models that perform voting locally rather than globally. I also plan to try adapting the hill-climbing algorithm described in Chapter 4 to produce multiple complementary encodings, by searching for a set of encodings that individually perform well, but that are substantially different from one another.

9.5 Non-Local Output Features

Chapter 8 presents a method for adding features that depend on non-local pieces of output structure to Bayesian Models. For my dissertation, I plan to evaluate the effect of this method, on both performance and run-time, by using augmentation variables to add manually selected classes of non-local features to existing state-of-the-art linear chain CRF systems. I also plan to combine this algorithm with the hill-climbing system that was described in Chapter 4. This should improve the hill-climbing system's performance, by allowing it to introduce new long-distance output dependencies without needing to split the training data for local sub-problems.

9.6 Proposed Schedule

Chunking: *February 2007 - September 2007*	
02/07 - 03/07	Adapt the hill-climbing algorithm to the Biomedical Named Entity Detection task. (Section 9.1)
04/07 - 06/07	Update the hill-climbing algorithm to make use of the method for adding non-local output features. (Section 9.5)
07/07 - 09/07	Experiments to provide the hill-climbing search algorithm with more direction. (Section 9.1)
Semantic Role Labeling: *October 2007 - February 2008*	
10/07 - 11/07	Experiments that evaluate the effect of transforming semantic role labels via SemLink. (Section 9.2).
12/07 - 02/08	Apply the hill-climbing algorithm to semantic role labeling. (Section 9.3).
Voting: *March 2008 - May 2008*	
03/08 - 05/08	Experiments to evaluate the global voting algorithm. (Section 9.4).
Dissertation: *May 2008 - June 2008*	
05/08 - 06/08	Write dissertation.
06/08	Dissertation defense.

Appendix A

Finding the Optimal Group Sequence is NP-Hard

As mentioned in Section 7.4, 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 [5], 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 $x_{i,j} \in (\{v_1, \ldots, v_n\} \cup \{\overline{v_1}, \ldots, \overline{v_n}\})$.

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

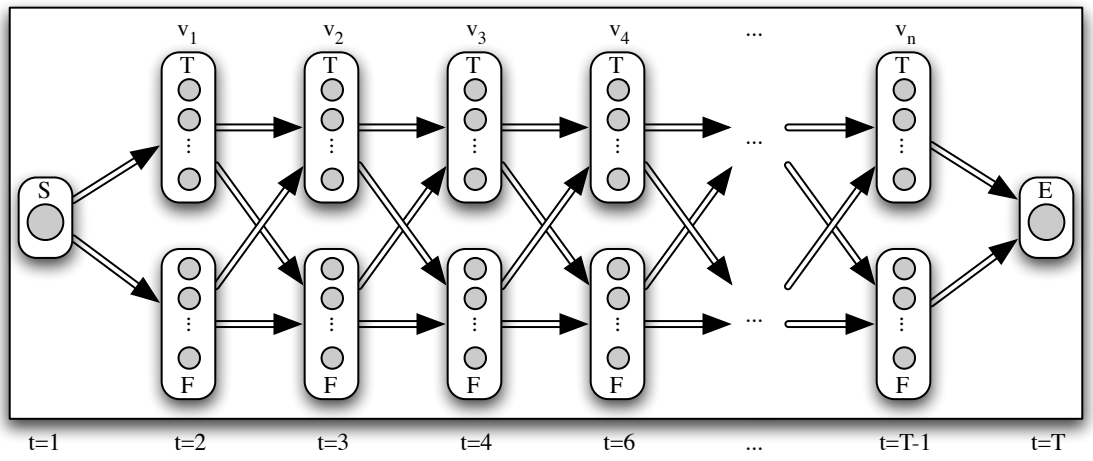


Figure A.1: **Basic Architecture of Graph Corresponding to 3-SAT**. Each variable v_i is represented by a single time slice $t = i + 1$. Two groups, "T," and "F," are used to represent the variables' truth values. Arrows represent paths through group nodes (as opposed to paths through individual subnodes).

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 highest scoring group path through the combined graph. If the score of this group

[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.

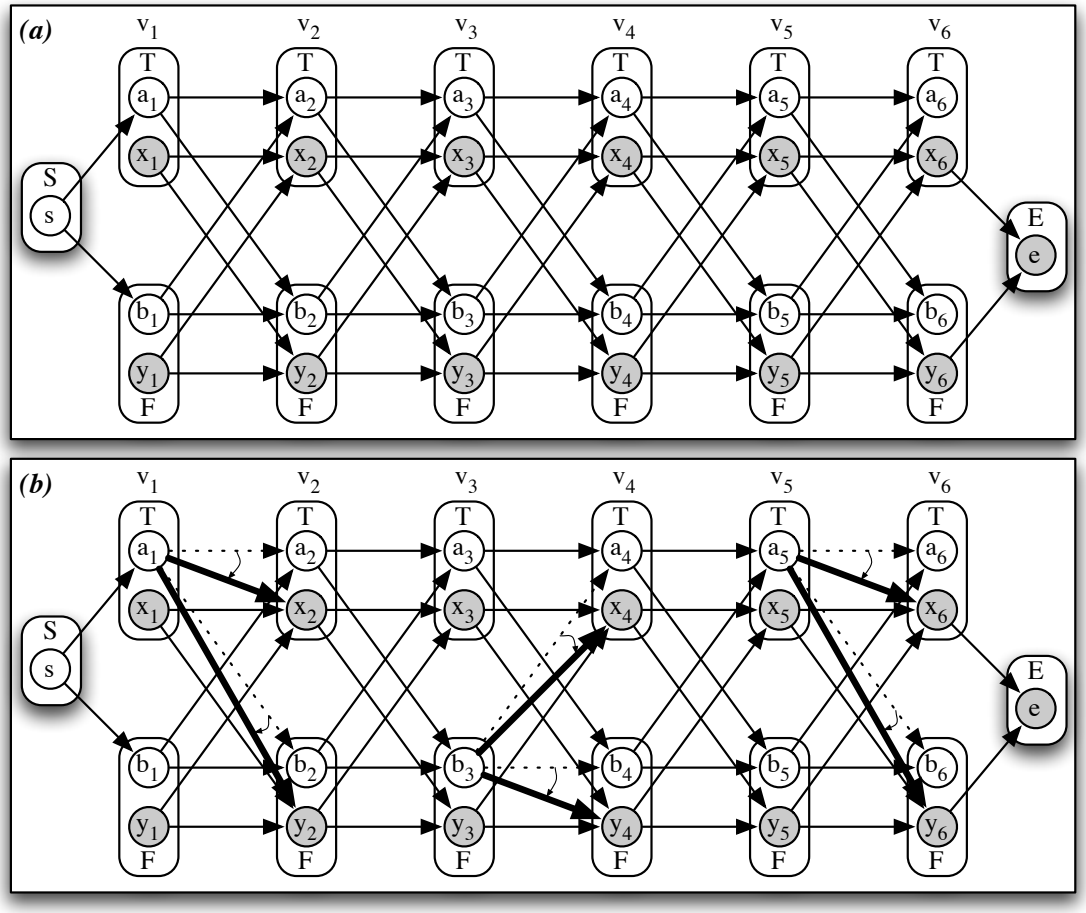


Figure A.2: **Construction of the Subgraph Corresponding to a Single Clause**. This figure shows how the subnode corresponding to the clause $(v_1 \vee \overline{v_3} \vee v_5)$ is constructed (with n=6 variables). In step *(a)*, we create two disconnected subgraphs. The first subgraph (unshaded nodes) 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.

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

[1] Ethem Alpaydin. *Introduction to Machine Learning*, chapter 7. The MIT Press, 2004.

[2] Daniel M. Bikel. Intricacies of collin's parsing model. *Computational Linguistics*, 30(4), 2004.

[3] Xavier Carreras and Lluís Márquez. Introduction to the conll-2004 shared task: Semantic role labeling. In *Proceedings of CoNLL*, 2004.

[4] Xavier Carreras and Lluís Márquez. Introduction to the conll-2005 shared task: Semantic role labeling. In *Proceedings of CoNLL*, 2005.

[5] F. Casacuberta and Colin De La Higuera. 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, 2000.

[6] John Chen and Owen Rambow. Use of deep linguistic features for the recognition and labeling of semantic arguments. In *Proceedings of EMNLP-2003*, Sapporo, Japan, 2003.

[7] Michael Collins. *Head-Driven Statistical Models for Natural Language Processing*. PhD thesis, University of Pennsylvania, 1999.

[8] Walter Daelemans, Jakub Zavrel, Ko van der Sloot, and Antal van den Bosch. Timbl: Tilburg memory based learner, version 2.0, reference guide. Technical Report 99-01, ILK, 1999.

[9] S. Dasgupta. Learning mixtures of gaussians. In *Fortieth Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, 1999.

[10] T. G. Dietterich and G. Bakiri. Solving multiclass learning problems via error-correcting output codes. *Journal of Artificial Intelligence Research*, 2:263–286, 1995.

[11] Christy Doran, Dania Egedi, Beth Ann Hockey, B. Srinivas, and Martin Zaidel. 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, 1994.

[12] D. Dowty. 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, 1989.

[13] D. Dowty. Thematic proto-roles and argument selection. *Language*, 67(3):547–619, 1991.

[14] Kai-Bo Duan and S. Sathiya Keerthi. Which is the best multiclass svm method? an empirical study. In *Multiple Classifier Systems*, pages 278–285. Springer Berlin / Heidelberg, 2005.

[15] C. J. Fillmore. The case for case. In E. Bach and R. Harms, editors, *Universals in Linguistic Theory*. Holt, Rinchart, and Winston, New York, 1968.

[16] Daniel Gildea and Julia Hockenmaier. Identifying semantic roles using Combinatory Categorial Grammar. In *2003 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 57–64, Sapporo, Japan, 2003.

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

[18] J. Gruber. *Studies in Lexical Relations*. PhD thesis, MIT, 1965.

[19] Kadri Hacioglu, Sameer Pradhan, Wayne Ward, James H. Martin, and Daniel Jurafsky. Shallow semantic parsing using support vector machines. Technical report, The Center for Spoken Language Research at the University of Colorado (CSLR), 2003.

[20] Chih-Wei Hsu and Chih-Jen Lin. A comparison of methods for multiclass support vector machines. *IEEE Transactions on Neural Networks*, 13(2):415–425, March 2002.

[21] R. Jackendoff. *Semantic Interpretation in Generative Grammar*. MIT Press, Cambridge, Massachusetts, 1972.

[22] Mark Johnson. Pcfg models of linguistic tree representations. *Computational Linguistics*, 24(4), 1998.

[23] Dan Klein and Christopher D. Manning. Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics*, 2003.

[24] Dan Klein and Christopher D. Manning. Factored a* search for models over sequences and trees. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, 2003.

[25] Dan Klein and Christopher D. Manning. 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, 2003. MIT Press.

[26] Taku Kudo and Yuji Matsumoto. Chunking with support vector machines. In *Proceedings of NAACL-2001.*, 2001.

[27] Beth Levin. *English Verb Classes and Alternations: A Preliminary Investigation.* The University of Chicago Press, 1993.

[28] Edward Loper, Szu ting Yi, and Martha Palmer. Combining lexical resources: Mapping between propbank and verbnet. In *Proceedings of the 7th International Workshop on Computational Linguistics*, Tilburg, the Netherlands, 2007.

[29] M. Marcus, G. Kim, M. Marcinkiewicz, R. MacIntyre, A. Bies, M. Ferguson, K. Katz, and B. Schasberger. The Penn treebank: Annotating predicate argument structure, 1994.

[30] Takuya Matsuzaki, Yusuke Miyoa, and Jun'ichi Tsujii. Probabilistic cfg with latent annotations. In *Proceedings of the 43rd Annual Meeting of the ACL*, 2005.

[31] Andrew McCallum, Dayne Freitag, and Fernando Pereira. 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, 2000.

[32] Andrew Kachites McCallum. Mallet: A machine learning for language toolkit. http://mallet.cs.umass.edu, 2002.

[33] Mehryar Mohri. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–311, 1997.

[34] Thomas Morton. *Using Reference Relations to Improve Information Retrieval.* PhD thesis, University of Pennsylvania, 2004.

[35] Alessandro Moschitti. A study on convolution kernel for shallow semantic parsing. In *Proceedings of the 42-th Conference on Association for Computational Linguistic (ACL-2004)*, Barcelona, Spain, 2004.

[36] Martha Palmer, Daniel Gildea, and Paul Kingsbury. The proposition bank: A corpus annotated with semantic roles. *Computational Linguistics*, 31(1):71–106, 2005.

[37] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann, 1988.

[38] Slav Petrov, Leon Barrett, Romain Thibaux, and Dan Klein. 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, 2006.

[39] S. Pradhan, W. Ward, K. Hacioglu, J. Martin, and D. Jurafsky. 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-2004)*, Boston, MA, 2004.

[40] Sameer Pradhan, Kadri Hacioglu, Wayne Ward, H. Martin, James, and Daniel Jurafsky. Semantic role chunking combining complementary syntactic views. In *Proceedings of CoNLL-2005*, 2005.

[41] Sameer Pradhan, Wayne Ward, Kadri Hacioglu, James Martin, and Dan Jurafsky. Semantic role labeling using different syntactic views. In *Proceedings of the Association for Computational Linguistics 43rd annual meeting (ACL-2005)*, Ann Arbor, MI, 2005.

[42] V. Punyakanok, D. Roth, and W. Yih. The necessity of syntactic parsing for semantic role labeling. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI-05)*, 2005.

[43] Lance Ramshaw and Mitch Marcus. 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, 1995. Association for Computational Linguistics.

[44] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.

[45] Erik F. Tjong Kim Sang. Noun phrase recognition by system combination. In *Proceedings of BNAIC*, Tilburg, The Netherlands, 2000.

[46] Karin Kipper Schuler. *VerbNet: A broad-coverage, comprehensive verb lexicon*. PhD thesis, University of Pennsylvania, 2005.

[47] Fei Sha and Fernando Pereira. Shallow parsing with conditional random fields. In *Proceedings of HLT-NAACL*, pages 134–141, 2003.

[48] Hong Shen and Anoop Sarkar. 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 2005.

[49] P. Smyth. Belief networks, hidden markov models, and markov random fields: a unifying view. *Pattern Recognition Letters*, 1998.

[50] Charles Sutton and Andrew McCallum. Joint parsing and semantic role labeling. In *Proceeding of the 9th Conference on Computational Natural Language Learning (CoNLL)*, pages 225–228, 2005.

[51] Charles Sutton and Andrew McCallum. An introduction to conditional random fields for relational learning. In Lise Getoor and Ben Taskar, editors, *Introduction to Statistical Relational Learning*. MIT Press, 2006. To appear.

[52] Cynthia A. Thompson, Roger Levy, and Christopher D. Manning. A generative model for semantic role labeling. In *Proceedings of ECML-2003*, 2003.

[53] Erik Tjong Kim Sang and Jorn Veenstra. Representing text chunks. In *Proceedings of EACL'99*, Bergen, 1999. Association for Computational Linguistics.

[54] Kristina Toutanova, Aria Haghighi, and Christopher D. Joint learning improves semantic role labeling. In *Proceedings of the Association for Computational Linguistics 43rd annual meeting (ACL-2005)*, Ann Arbor, MI, 2005.

[55] Szu-ting Yi, Edward Loper, and Martha Palmer. 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-2007)*, 2007.

[56] Szu-ting Yi and Martha Palmer. Pushing the boundaries of semantic role labeling with svm. In *Proceedings of the International Conference on Natural Language Processing*, 2004.

[57] Szu-ting Yi and Martha Palmer. The integration of syntactic parsing and semantic role labeling. In *Proceedings of CoNLL-2005*, 2005.