A Technical Introduction to Statistical Natural Language Processing

Jacob Eisenstein

December 23, 2016

Contents

[1] This section follows closely from J&M 2009

Preface

This text is built from the notes that I use for teaching Georgia Tech's undergraduate and graduate courses on natural language processing, CS 4650 and 7650. There are several other good resources (e.g., Manning and Schütze, 1999; Jurafsky and Martin, 2009; Smith, 2011; Figueiredo et al., 2013; Collins, 2013), but for various reasons I wanted to create something of my own.

The text assumes familiarity with basic linear algebra, and with calculus through Lagrange multipliers. It includes a refresher on probability, but some previous exposure would be helpful. An introductory course on the analysis of algorithms is also assumed; in particular, the reader should be familiar with asymptotic analysis of the time and memory costs of algorithms, and should have seen dynamic programming. No prior background in machine learning or linguistics is assumed, and even students with background in machine learning should be sure to read the introductory chapters, since the notation used in natural language processing is different from typical machine learning presentations, due to the emphasis on structure prediction in applications of machine learning to language. Throughout the book, advanced material is marked with an asterisk, and can be safely skipped.

The notes focus on what I view as a core subset of the field of natural language processing, unified by the concepts of linear models and structure prediction. A remarkable thing about the field of natural language processing is that so many problems in language technology can be solved by a small number of methods. These notes focus on the following methods:

Search algorithms shortest path, Viterbi, CKY, minimum spanning tree, shift-reduce, integer linear programming, dual decomposition (maybe), beam search.

Learning algorithms Naïve Bayes, logistic regression, perceptron, expectation-maximization, matrix factorization, backpropagation.

The goal of this text is to teach how these methods work, and how they can be applied to problems that arise in the computer processing of natural language: document classification, word sense disambiguation, sequence labeling (part-of-speech tagging and named entity recognition), parsing, coreference resolution, relation extraction, discourse analysis, and, to a limited degree, language modeling and machine translation. Because proper application of these techniques requires understanding the underlying linguistic phenomena, the notes also include chapters on the foundations of morphology, syntactic parts of speech, context-free grammar, semantics, and discourse; however, for a detailed understanding of these topics, a full-fledged linguistics textbook should be consulted (e.g., Akmajian et al., 2010; Fromkin et al., 2013).

-Jacob Eisenstein, December 23, 2016

Notation

w_n	word token at position n
$\boldsymbol{x}^{(i)}$	a (column) vector of feature counts for instance i, often word counts
$\boldsymbol{x}_{i:j}$	elements i through j (inclusive) of a vector $\boldsymbol{x}$
$y^{(i)}$	the label for instance i
$\hat{y}$	a predicted label
$\boldsymbol{y}$	a vector of labels across all instances
$\mathcal{Y}$	the set of all possible labels
K	number of possible labels $K = \#\lvert\mathcal{Y}\rvert$
$\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)})$	feature vector for instance i with label $y^{(i)}$
$\boldsymbol{\theta}$	a (column) vector of weights
$\boldsymbol{\theta}^{\top}$	the transpose of the vector $\boldsymbol{\theta}$
$\Pr(A)$	probability of event A
$\mathrm{p}_B(b)$	the marginal probability of random variable B taking value b
$\mathcal{T}(\boldsymbol{w})$	the set of possible tag sequences for the word sequence $\boldsymbol{w}$
$\lozenge$	the start tag
$\blacklozenge$	the stop tag
$\square$	the start token
$\blacksquare$	the stop token
λ	the amount of regularization

Part I

Words, bags of words, and features

Chapter 1

Linear classification and features

Suppose you want to build a spam detector, in which each document is classified as "spam" or "ham." How would you do it, using only the text in the email?

One solution is to represent document i as a column vector of word counts: $\boldsymbol{x}^{(i)} = [0\ 1\ 1\ 0\ 0\ 2\ 0\ 1\ 13\ 0 \ldots]^{\top}$, where $x_{i,j}$ is the count of word j in document i. Suppose the size of the vocabulary is V, so that the length of $\boldsymbol{x}^{(i)}$ is also V. The object $\boldsymbol{x}^{(i)}$ is a vector, but colloquially we call it a **bag of words**, because it includes only information about the count of each word, and not the order in which they appear.

We've thrown out grammar, sentence boundaries, paragraphs — everything but the words! But this could still work. If you see the word *free*, is it spam or ham? How about *Bayesian*? One approach would be to define a "spamminess" score for every word in the dictionary, and then just add them up. These scores are called **weights**, written $\boldsymbol{\theta}$, and we'll spend a lot of time talking about where they come from.

But for now, let's generalize: suppose we want to build a multi-way classifier to distinguish stories about sports, celebrities, music, and business. Each label $y^{(i)}$ is a member of a set of K possible labels $\mathcal{Y}$. Our goal is to predict a label $\hat{y}^{(i)}$, given the bag of words $\boldsymbol{x}^{(i)}$, using the weights $\boldsymbol{\theta}$. We'll do this using a vector inner product between the weights $\boldsymbol{\theta}$ and a **feature vector** $\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)})$. As the notation suggests, the feature vector is constructed by combining $\boldsymbol{x}^{(i)}$ and $y^{(i)}$. For example, feature j might be,

$$f_j(\boldsymbol{x}^{(i)}, y^{(i)}) = \begin{cases} 1, & \text{if}(\textit{freeee} \in \boldsymbol{x}^{(i)}) \wedge (y^{(i)} = \textsc{Spam}) \\ 0, & \text{otherwise} \end{cases} \tag{1.1}$$

For any pair $\langle \boldsymbol{x}^{(i)}, y^{(i)} \rangle$, we then define $\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)})$ as,

$$\boldsymbol{f}(\boldsymbol{x}, Y = 0) = [\boldsymbol{x}^\top, \underbrace{0, 0, \ldots, 0}_{V \times (K-1)}]^\top \tag{1.2}$$

$$\boldsymbol{f}(\boldsymbol{x}, Y = 1) = [\underbrace{0, 0, \ldots, 0}_{V}, \boldsymbol{x}^\top, \underbrace{0, 0, \ldots, 0}_{V \times (K-2)}]^\top \tag{1.3}$$

$$\boldsymbol{f}(\boldsymbol{x}, Y = 2) = [\underbrace{0, 0, \ldots, 0}_{2 \times V}, \boldsymbol{x}^\top, \underbrace{0, 0, \ldots, 0}_{V \times (K-3)}]^\top \tag{1.4}$$

$$\boldsymbol{f}(\boldsymbol{x}, Y = K) = [\underbrace{0, 0, \ldots, 0}_{V \times (K-1)}, \boldsymbol{x}^\top]^\top, \tag{1.5}$$

where $\underbrace{0, 0, \ldots, 0}_{V \times (K-1)}$ is a column vector of $V \times (K-1)$ zeros. This arrangement is shown in Figure 1.1. This notation may seem like a strange choice, but in fact it helps to keep things simple. Given a vector of weights, $\boldsymbol{\theta} \in \mathbb{R}^{V \times K}$, we can now compute the inner product $\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}, y)$. This inner product gives a scalar measure of the score for label y, given observations $\boldsymbol{x}$. For any document $\boldsymbol{x}^{(i)}$, we predict the label $\hat{y}$ as

$$\hat{y} = \underset{y}{\operatorname{argmax}}\, \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y) \tag{1.6}$$

This inner product is the fundamental equation for linear classification, and it is the reason we prefer the feature function notation $\boldsymbol{f}(\boldsymbol{x}, y)$. The notation gives a clean separation between the **data** ($\boldsymbol{x}$ and y) and the **parameters**, which are expressed by the single vector of weights, $\boldsymbol{\theta}$. As we will see in later chapters, this notation also generalizes nicely to **structured output spaces**, in which the space of labels $\mathcal{Y}$ is very large, and we want to model shared substructure between labels.

Often we'll add an **offset** feature at the end of $\boldsymbol{x}$, which is always 1; we then have to also add an extra zero to each of the zero vectors. This gives the entire feature vector $\boldsymbol{f}(\boldsymbol{x}, y)$ a length of $(V+1) \times K$. The weight associated with this offset feature can be thought of as a "bias" for each label. For example, if we expect most documents to be spam, then the weight for the offset feature for $Y = \text{spam}$ should be larger than the weight for the offset feature for $Y = \text{ham}$.

Returning to the weights $\boldsymbol{\theta}$ — where do they come from? As already suggested, we could set the weights by hand. If we wanted to distinguish, say, English from Spanish, we could use English and Spanish dictionaries, and set the weight to one for each word that appears in the associated dictionary. For exam-

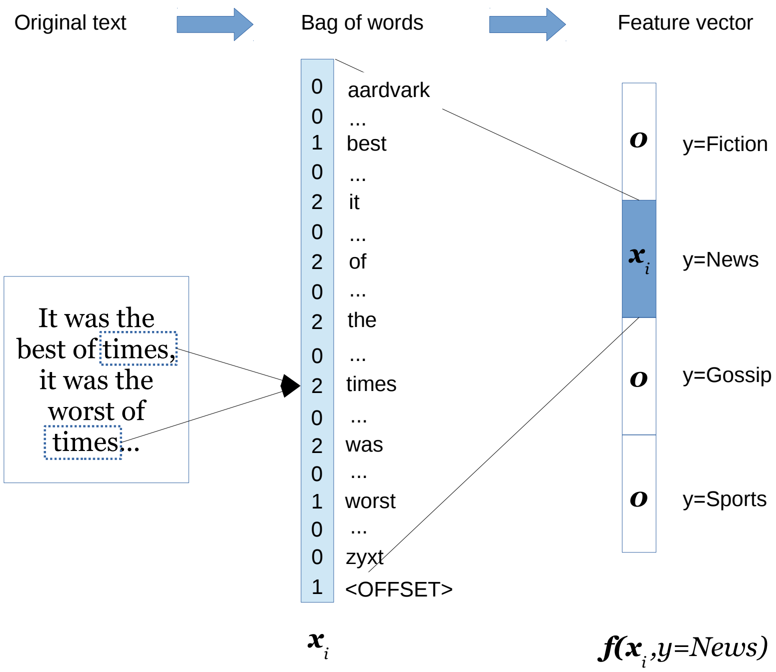


Figure 1.1: The bag-of-words and feature vector representations, for a hypothetical text classification task.

ple,[1]

$$\begin{aligned}
\theta_{(E,bicycle)} =& 1 & \theta_{(S,bicycle)} =& 0\\
\theta_{(E,bicicleta)} =& 0 & \theta_{(S,bicicleta)} =& 1\\
\theta_{(E,con)} =& 1 & \theta_{(S,con)} =& 1\\
\theta_{(E,ordinateur)} =& 0 & \theta_{(S,ordinateur)} =& 0.
\end{aligned}$$

Similarly, if we want to distinguish positive and negative sentiment, we could use positive and negative *sentiment lexicons*, which are defined by expert psychologists (Tausczik and Pennebaker, 2010). You'll try this in Problem Set 1.

But it is usually not easy to set classification weights by hand. Instead, we will learn them from data. For example, email users manually label thousands of messages as "spam" or "not spam"; newspapers label their own articles as "business" or "fashion." Such **instance labels** are a typical form of labeled data that we will encounter in NLP. In **supervised machine learning**, we use instance

[1] In this notation, each tuple (language, word) indexes an element in $\boldsymbol{\theta}$, which remains a vector.

labels to automatically set the weights for a classifier. An important tool for this is probability.

1.1 Review of basic probability

Probability theory provides a way to reason about random events. The sorts of random events that are typically used to explain probability theory include coin flips, card draws, and the weather. It may seem odd to think about the choice of a word as akin to the flip of a coin, particularly if you are the type of person to choose words carefully. But random or not, language has proven to be extremely difficult to model deterministically. Probability offers a powerful tool for modeling and manipulating linguistic data, which we will use repeatedly throughout this course.[2]

Probability can be thought of in terms of **random outcomes**: for example, a single coin flip has two possible outcomes, heads or tails. The set of possible outcomes is the **sample space**, and a subset of the **sample space** is an **event**. For a sequence of two coin flips, there are four possible outcomes, $\{HH, HT, TH, TT\}$, representing the ordered sequences heads-head, heads-tails, tails-heads, and tails-tails. The event of getting exactly one head includes two outcomes: $\{HT, TH\}$.

Formally, a probability is a function from events to the interval between zero and one: $\Pr : \mathcal{F} \to [0, 1]$, where $\mathcal{F}$ is the set of possible events. An event that is certain has probability one; an event that is impossible has probability zero. For example, the probability of getting less than three heads on two coin flips is one. Each outcome is also an event (a set with exactly one element), and for two flips of a fair coin, the probability of each outcome is,

$$\Pr(\{HH\}) = \Pr(\{HT\}) = \Pr(\{TH\}) = \Pr(\{TT\}) = \frac{1}{4}. \tag{1.7}$$

Probabilities of event combinations

Because events are **sets** of outcomes, we can use set theoretic operations such complement, intersection, and unions to reason about the probabilities of various event combinations.

For any event A, there is a **complement** $\neg A$, such that:

[2] A good introduction to probability theory is offered by Manning and Schütze (1999), which helped to motivate this section. For more detail, Sharon Goldwater provides another useful reference, `http://homepages.inf.ed.ac.uk/sgwater/teaching/general/probability.pdf`.

- The union $A \cup \neg A$ covers the entire sample space, and $\Pr(A \cup \neg A) = 1$;
- The intersection $A \cap \neg A = \varnothing$ is the empty set, and $\Pr(A \cap \neg A) = 0$.

In the coin flip example, the event of obtaining a single head on two flips corresponds to the set of outcomes $\{HT, TH\}$; the complement event includes the other two outcomes, $\{TT, HH\}$.

Probabilities of disjoint events

In general, when two events have an empty intersection, $A \cap B = \varnothing$, they are said to be **disjoint**. The probability of the union of two disjoint events is equal to the sum of their probabilities,

$$A \cap B = \varnothing \quad \Rightarrow \quad \Pr(A \cup B) = \Pr(A) + \Pr(B). \tag{1.8}$$

This is the **third axiom of probability**, and can be generalized to any countable sequence of disjoint events.

In the coin flip example, we can use this axiom to derive the probability of the event of getting a single head on two flips. This event is the set of outcomes $\{HT, TH\}$, which is the union of two simpler events, $\{HT, TH\} = \{HT\} \cup \{TH\}$. The events $\{HT\}$ and $\{TH\}$ are disjoint. Therefore,

$$\begin{aligned}\Pr(\{HT, TH\}) = \Pr(\{HT\} \cup \{TH\}) =& \Pr(\{HT\}) + \Pr(\{TH\}) && (1.9)\\ =& \frac{1}{4} + \frac{1}{4} = \frac{1}{2}. && (1.10)\end{aligned}$$

For events that are not disjoint, it is still possible to compute the probability of their union:

$$\Pr(A \cup B) = \Pr(A) + \Pr(B) - \Pr(A \cap B). \tag{1.11}$$

This can be derived from the third axiom of probability. First, consider an event that includes all outcomes in B that are not in A, which we can write as $B - (A \cap B)$. By construction, this event is disjoint from A.[3] We can therefore apply the additive rule,

$$\begin{aligned}\Pr(A \cup B) =& \Pr(A) + \Pr(B - (A \cap B)) && (1.12)\\ \Pr(B) =& \Pr(B - (A \cap B)) + \Pr(A \cap B) && (1.13)\\ \Pr(B - (A \cap B)) =& \Pr(B) - \Pr(A \cap B) && (1.14)\\ \Pr(A \cup B) =& \Pr(A) + \Pr(B) - \Pr(A \cap B). && (1.15)\end{aligned}$$

[3] [todo: add figure]

Law of total probability

A set of events $\mathcal{B} = \{B_1, B_2, \ldots, B_N\}$ is a **partition** of the sample space iff each pair of events is disjoint ($B_i \cap B_j = \varnothing$), and the union of the events is the entire sample space. The law of total probability states that we can **marginalize** over these events as follows,

$$\Pr(A) = \sum_{B_n \in \mathcal{B}} \Pr(A \cap B_n). \tag{1.16}$$

Note for any event B, the union $B \cup \neg B$ forms a partition of the sample space. Therefore, an important special case of the law of total probability is,

$$\Pr(A) = \Pr(A \cap B) + \Pr(A \cap \neg B). \tag{1.17}$$

Conditional probability and Bayes' rule

A **conditional probability** is an expression like $\Pr(A \mid B)$, which is the probability of the event A, assuming that event B happens too. For example, we may be interested in the probability of a randomly selected person answering the phone by saying *hello*, conditioned on that person being a speaker of English. We define conditional probability as the ratio,

$$\Pr(A \mid B) = \frac{\Pr(A \cap B)}{\Pr(B)} \tag{1.18}$$

The **chain rule** states that $\Pr(A \cap B) = \Pr(A \mid B) \times \Pr(B)$, which is just a simple rearrangement of terms from Equation 1.18. We can apply the chain rule repeatedly:

$$\begin{aligned}\Pr(A \cap B \cap C) =& \Pr(A \mid B \cap C) \times \Pr(B \cap C)\\ =& \Pr(A \mid B \cap C) \times \Pr(B \mid C) \times \Pr(C)\end{aligned}$$

Bayes' rule (sometimes called Bayes' law or Bayes' theorem) gives us a way to convert between $\Pr(A \mid B)$ and $\Pr(B \mid A)$. It follows from the chain rule:

$$\Pr(A \mid B) = \frac{\Pr(A \cap B)}{\Pr(B)} = \frac{\Pr(B \mid A) \times \Pr(A)}{\Pr(B)} \tag{1.19}$$

The terms in Bayes rule have specialized names, which we will occasionally use:

- $\Pr(A)$ is the **prior**, since it is the probability of event A without knowledge about whether B happens or not.
- $\Pr(B \mid A)$ is the **likelihood**, the probability of event B given that event A has occurred.
- $\Pr(A \mid B)$ is the **posterior**, since it is the probability of event A with knowledge that B has occurred.

Example Manning and Schütze (1999) have a nice example of Bayes' rule (sometimes called Bayes Law) in a linguistic setting. (This same example is usually framed in terms of tests for rare diseases.) Suppose one is interested in a rare syntactic construction, such as **parasitic gaps**, which occur on average once in 100,000 sentences. Here is an example:

(1.1) *Which class did you attend __ without registering for __?*

Lana Linguist has developed a complicated pattern matcher that attempts to identify sentences with parasitic gaps. It's pretty good, but it's not perfect:

- If a sentence has a parasitic gap, the pattern matcher will find it with probability 0.95. (Skipping ahead, this is the **recall**; the **false negative rate** is defined as one minus the recall.)
- If the sentence doesn't have a parasitic gap, the pattern matcher will wrongly say it does with probability 0.005. (This is the **false positive rate**. The **precision** is defined as one minus the false positive rate.)

Suppose that Lana's pattern matcher says that a sentence contains a parasitic gap. What is the probability that this is true?

Let G be the event of a sentence having a parasitic gap, and T be the event of the test being positive. We are interested in the probability of a sentence having a parasitic gap given that the test is positive. This is the conditional probability $\Pr(G \mid T)$, and we can compute it from Bayes' rule:

$$\Pr(G \mid T) = \frac{\Pr(T \mid G) \times \Pr(G)}{\Pr(T)}. \tag{1.20}$$

We already know both terms in the numerator: $\Pr(T \mid G)$ is the recall, which is 0.95; $\Pr(G)$ is the prior, which is 10^{-5}.

We are not given the denominator, but we can compute it by using some of the tools that we have developed in this section. We first apply the law of total probability, using the partition $\{G, \neg G\}$:

$$\Pr(T) = \Pr(T \cap G) + \Pr(T \cap \neg G). \tag{1.21}$$

This says that the probability of the test being positive is the sum of the probability of a **true positive** ($T \cap G$) and the probability of a **false positive** ($T \cap \neg G$). Next, we can compute the probability of each of these events using the chain rule:

$$\begin{aligned}
\Pr(T \cap G) =& \Pr(T \mid G) \times \Pr(G) = 0.95 \times 10^{-5} && (1.22)\\
\Pr(T \cap \neg G) =& \Pr(T \mid G) \times \Pr(\neg G) = 0.005 \times (1 - 10^{-5}) \approx 0.005 && (1.23)\\
\Pr(T) =& \Pr(T \cap G) + \Pr(T \cap \neg G) && (1.24)\\
=& 0.95 \times 10^{-5} + 0.005 \approx 0.005. && (1.25)
\end{aligned}$$

We now return to Bayes' rule to compute the desired posterior probability,

$$\begin{aligned}
\Pr(G \mid T) =& \frac{\Pr(T \mid G) \Pr(G)}{\Pr(T)} && (1.26)\\
=& \frac{0.95 \times 10^{-5}}{0.95 \times 10^{-5} + 0.005 \times (1 - 10^{-5})} && (1.27)\\
\approx& 0.002. && (1.28)
\end{aligned}$$

Lana's pattern matcher is very accurate, with false positive and false negative rates below 5%. Yet the extreme rarity of this phenomenon means that a positive result from the detector is most likely to be wrong.

Independence

Two events are independent if the probability of their intersection is equal to the product of their probabilities: $\Pr(A \cap B) = \Pr(A) \times \Pr(B)$. For example, for two flips of a fair coin, the probability of getting heads on the first flip is independent of the probability of getting heads on the second flip. We can prove this by using the additive axiom defined above:

$$\begin{aligned}
\Pr(\{HT, HH\}) =& \Pr(HT) + \Pr(HH) = \frac{1}{4} + \frac{1}{4} = \frac{1}{2} && (1.29)\\
\Pr(\{HH, TH\}) =& \Pr(HH) + \Pr(TH) = \frac{1}{4} + \frac{1}{4} = \frac{1}{2} && (1.30)\\
\Pr(\{HT, HH\}) \times \Pr(\{HH, TH\}) =& \frac{1}{2} \times \frac{1}{2} = \frac{1}{4} && (1.31)\\
\Pr(\{HT, HH\} \cap \{HH, TH\}) =& \Pr(HH) = \frac{1}{4} && (1.32)\\
=& \Pr(\{HT, HH\}) \times \Pr(\{HH, TH\}). && (1.33)
\end{aligned}$$

Independence will play a key role in the discussion of probabilistic classification later in this chapter.

If $\Pr(A \cap B \mid C) = \Pr(A \mid C) \times \Pr(B \mid C)$, then the events A and B are **conditionally independent**, written $A \perp B \mid C$.

Random variables

Random variables are functions of events. Formally, we will treat random variables as functions from events to the space $\mathbb{R}^n$, where $\mathbb{R}$ is the set of real numbers. This general notion subsumes a number of different types of random variables:

- **Indicator random variables** are functions from events to the set $\{0, 1\}$. In the coin flip example, we can define X as an indicator random variable, for whether the coin has come up heads on at least one flip. This would include the outcomes $\{HH, HT, TH\}$. The event probability $\Pr(X = 1)$ is the sum of the probabilities of these outcomes, $\Pr(X = 1) = \frac{1}{4} + \frac{1}{4} + \frac{1}{4} = \frac{3}{4}$.
- A **discrete random variable** is a function from events to a countable subset of $\mathbb{R}$. Consider the coin flip example: the number of heads, X, can be viewed as a discrete random variable, $X \in 0, 1, 2$. The event probability $\Pr(X = 1)$ can again be computed as the sum of the probabilities of the events in which there is one head, $\{HT, TH\}$, giving $\Pr(X = 1) = \frac{1}{2}$.

Each possible value of a random variable is associated with a subset of the sample space. In the coin flip example, $X = 0$ is associated with the event $\{TT\}$, $X = 1$ is associated with the event $\{HT, TH\}$, and $X = 2$ is associated with the event $\{HH\}$. Assuming a fair coin, the probabilities of these events are, respectively, $1/4$, $1/2$, and $1/4$. This list of numbers represents the **probability distribution** over X, written p_X, which maps from the possible values of X to the non-negative reals. For a specific value x, we write $\mathrm{p}_X(x)$, which is equal to the event probability $\Pr(X = x)$.[4] The function p_X is called a probability **mass** function (pmf) if X is discrete; it is called a probability **density** function (pdf) if X is continuous. In either case, we have $\int_x \mathrm{p}_X(x) dx = 1$ and $\forall x, \mathrm{p}_X(x) \geq 0$.

Random variables can be combined into **joint probabilities**, e.g., $\mathrm{p}_{A,B}(a, b) = \Pr(A = a \cap B = b)$. Several ideas from event probabilities carry over to probability distributions over random variables:

- We can write a **marginal probability distribution** $\mathrm{p}_A(a) = \sum_b \mathrm{p}_{A,B}(a, b)$.

[4] In general, capital letters (e.g., X) refer to random variables, and lower-case letteres (e.g., x) refer to specific values. I will often just write $\mathrm{p}(x)$, when the subscript is clear from context.

- We can write a **conditional probability distribution** as $\text{p}_{A|B}(a \mid b) = \frac{\text{p}_{A,B}(a,b)}{\text{p}_B(b)}$.
- Random variables A and B are independent iff $\text{p}_{A,B}(a,b) = \text{p}_A(a) \times \text{p}_B(b)$.

Expectations

Sometimes we want the **expectation** of a function, such as $E[g(x)] = \sum_{x \in \mathcal{X}} g(x)\text{p}(x)$. Expectations are easiest to think about in terms of probability distributions over discrete events:

- If it is sunny, Marcia will eat three ice creams.
- If it is rainy, she will eat only one ice cream.
- There's a 80% chance it will be sunny.
- The expected number of ice creams she will eat is $0.8 \times 3 + 0.2 \times 1 = 2.6$.

If the random variable X is continuous, the sum becomes an integral:

$$E[g(x)] = \int_{\mathcal{X}} g(x)\text{p}(x)dx \tag{1.34}$$

For example, a fast food restaurant in Quebec gives a 1% discount on poutine[5] for every degree below zero. Assuming they use a thermometer with infinite precision, the expected price would be an integral over all possible temperatures,

$$E[\text{price}(x)] = \int_{\mathcal{X}} \min(1, 1+x) \times \text{original-price} \times \text{p}(x)dx. \tag{1.35}$$

(Careful readers will note that the restaurant will apparently pay you for taking poutine, if the temperature falls below -100 degrees celsius.)

1.2 Naïve Bayes

Back to text classification, where we were left wondering how to set the weights $\boldsymbol{\theta}$. Having just reviewed basic probability, we can now take a probabilistic approach to this problem. A Naïve Bayes classifier chooses the weights $\boldsymbol{\theta}$ to maximize the *joint* probability of a labeled dataset, $\text{p}(\{\boldsymbol{x}^{(i)}, y^{(i)}\}_{i \in 1 \dots N})$, where each tuple $\langle \boldsymbol{x}^{(i)}, y^{(i)} \rangle$ is a labeled instance.

[5] Readers from New Jersey will recognize poutine as a close relative of "disco fries."

We first need to define the probability $\mathrm{p}(\{\boldsymbol{x}^{(i)}, y^{(i)}\}_{i \in 1 \ldots N})$. We'll do that through a "generative model," which describes a hypothesized stochastic process that has generated the observed data.[6]

- For each document i,
  - draw the label $y^{(i)} \sim \text{Categorical}(\boldsymbol{\mu})$
  - draw the vector of counts $\boldsymbol{x}^{(i)} \mid y^{(i)} \sim \text{Multinomial}(\boldsymbol{\phi}_{y^{(i)}})$,

The first line of this generative model is "for each document i", which tells us to treat each document independently: the probability of the whole dataset is equal to the product of the probabilities of each individual document. The observed word counts and document labels are **independent and identically distributed (IID)**.

$$\mathrm{p}(\{\boldsymbol{x}^{(i)}, y^{(i)}\}_{i \in 1 \ldots N}; \boldsymbol{\mu}, \boldsymbol{\phi}) = \prod_{i=1}^{N} \mathrm{p}(\boldsymbol{x}^{(i)}, y^{(i)}; \boldsymbol{\mu}, \boldsymbol{\phi}) \tag{1.36}$$

This means that the words in each document are **conditionally independent** given the parameters $\boldsymbol{\mu}$ and $\boldsymbol{\phi}$.

The second line indicates $y^{(i)} \sim \text{Categorical}(\boldsymbol{\mu})$, which means that the random variable $y^{(i)}$ is a stochastic draw from a categorical distribution with **parameter $\boldsymbol{\mu}$**. A categorical distribution is just like a weighted die: $\mathrm{p}_{\text{cat}}(y; \boldsymbol{\mu}) = \mu_y$, where μ_y is the probability of the outcome $Y = y$. For example, if $\mathcal{Y} = \{\text{positive}, \text{negative}, \text{neutral}\}$, we might have $\boldsymbol{\mu} = [0.1, 0.7, 0.2]$. We require $\sum_y \mu_y = 1$ and $\forall_y, \mu_y \geq 0$.

The third and final line invokes the **multinomial distribution**, which is only slightly more complex:

$$\mathrm{p}_{\text{mult}}(\boldsymbol{x}; \boldsymbol{\phi}) = \frac{\left(\sum_j^V x_j\right)!}{\prod_j^V x_j!} \prod_j^V \phi_j^{x_j} \tag{1.37}$$

We again require that $\sum_j^V \phi_j = 1$ and $\forall j, \phi_j \geq 0$. The second part of the equation is a product over words, with an exponent for each word; recall that $\phi_j^0 = 1$

[6] We'll see a lot of different generative models in this course. They are a helpful tool because they clearly and explicitly define the assumptions that underly the form of the probability distribution. For a very readable introduction to generative models in statistics, see Blei (2014).

for all ϕ_j; this means that the words that have zero count play no role in the overall probability.

The first part of Equation 1.37 doesn't depend on $\boldsymbol{\phi}$, and can usually be ignored. Can you see why we need the first part at all?[7] We will return to this issue shortly.

We can write $\mathrm{p}(\boldsymbol{x}^{(i)} \mid y^{(i)}; \boldsymbol{\phi})$ to indicate the conditional probability of word counts $\boldsymbol{x}^{(i)}$ given label $y^{(i)}$, with parameter $\boldsymbol{\phi}$, which is equal to $\mathrm{p}_{\text{mult}}(\boldsymbol{x}^{(i)}; \boldsymbol{\phi}_{y^{(i)}})$. By specifying the multinomial distribution, we are working with *multinomial naïve Bayes* (MNB). Why "naïve"? Because the multinomial distribution treats each word token independently: the probability mass function factorizes across the counts.[8] We'll see this more clearly later, when we show how MNB is an example of linear classification.

Another version of Naïve Bayes

Consider a slight modification to the generative story of NB:

- For each document i
  - Draw the label $y^{(i)} \sim \text{Categorical}(\boldsymbol{\mu})$
  - For each word $n \leq D_i$
    - Draw the word $w_{i,n} \sim \text{Categorical}(\boldsymbol{\phi}_{y^{(i)}})$

This is not quite the same model as multinomial Naive Bayes (MNB): it's a product of categorical distributions over words, instead of a multinomial distribution over word counts. This means we would generate the words in order, like p_W(*multinomial*)p_W(*Naive*)p_W(*Bayes*). Formally, this is a model for the joint probability $\mathrm{p}(\boldsymbol{w}, y)$, not $\mathrm{p}(\boldsymbol{x}, y)$.

However, as a classifier, it is identical to MNB. The final probabilities are reduced by a factor corresponding to the normalization term in the multinomial, $\frac{(\sum_j x_j)!}{\prod_j x_j!}$. This means that the the probability for a vector of counts $\boldsymbol{x}$ is larger

[7] Technically, a multinomial distribution requires a second parameter, the total number of counts, which in the bag-of-words representation is equal to the number of words in the document.

[8] You can plug in any probability distribution to the generative story and it will still be naïve Bayes, as long as you are making the "naïve" assumption that your features are conditionally independent, given the label. For example, a multivariate Gaussian with diagonal covariance would be naïve in exactly the same sense.

than the probability for a list of words $\boldsymbol{w}$ that induces the same counts. But this makes sense: there can be many word sequences that correspond to a single vector counts. For example, *man bites dog* and *dog bites man* correspond to an identical count vector, $\{\textit{bites} : 1, \textit{dog} : 1, \textit{man} : 1\}$, and the total number of word orderings for a given count vector $\boldsymbol{x}$ is exactly the ratio $\frac{(\sum_j x_j)!}{\prod_j x_j!}$.

From the perspective of classification, none of this matters, because it has nothing to do with the label y or the parameters $\boldsymbol{\phi}$. The ratio of probabilities between any two labels y_1 and y_2 will be identical in the two models, as will the maximum likelihood estimates for the parameters $\boldsymbol{\mu}$ and $\boldsymbol{\phi}$ (defined later).

Prediction

The Naive Bayes prediction rule is to choose the label y which maximizes $\text{p}(\boldsymbol{x}, y; \boldsymbol{\mu}, \boldsymbol{\phi})$:

$$\begin{aligned}
\hat{y} =& \underset{y}{\operatorname{argmax}}\, \text{p}(\boldsymbol{x}, y; \boldsymbol{\mu}, \boldsymbol{\phi}) && (1.38)\\
=& \underset{y}{\operatorname{argmax}}\, \text{p}(\boldsymbol{x} \mid y; \boldsymbol{\phi})\text{p}(y; \boldsymbol{\mu}) && (1.39)\\
=& \underset{y}{\operatorname{argmax}} \log \text{p}(\boldsymbol{x} \mid y; \boldsymbol{\phi}) + \log \text{p}(y; \boldsymbol{\mu}) && (1.40)
\end{aligned}$$

Converting to logarithms makes the notation easier. It doesn't change the prediction rule because the log function is monotonically increasing.

Now we can plug in the probability distributions from the generative story.

$$\begin{aligned}
\log \text{p}(\boldsymbol{x} \mid y; \boldsymbol{\phi}) + \log \text{p}(y; \boldsymbol{\mu}) =& \log \left[\frac{\left(\sum_j x_j\right)!}{\prod_j x_j!} \prod_j \boldsymbol{\phi}_{y,j}^{x_j} \right] + \log \mu_y && (1.41)\\
=& \log \frac{\left(\sum_j x_j\right)!}{\prod_j x_j!} + \sum_j x_j \log \phi_{y,j} + \log \mu_y && (1.42)\\
=& k + \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}, y), && (1.43)
\end{aligned}$$

where

$$\begin{aligned}
\boldsymbol{\theta} =& [\boldsymbol{\theta}^{(1)\top}, \boldsymbol{\theta}^{(2)\top}, \ldots, \boldsymbol{\theta}^{(K)\top}]^\top && (1.44)\\
\boldsymbol{\theta}^{(y)} =& [\log \phi_{y,1},\ \log \phi_{y,2},\ \ldots,\ \log \phi_{y,V},\ \log \mu_y]^\top && (1.45)\\
k =& \log \frac{\left(\sum_j x_j\right)!}{\prod_j x_j!} \qquad \text{(This is a constant that we can ignore.)} && (1.46)
\end{aligned}$$

The feature function $\boldsymbol{f}(\boldsymbol{x}, y)$ is a vector of V word counts and an offset, padded by zeros for the labels not equal to y (see equations 1.2-1.5, and Figure 1.1). This construction ensures that the inner product $\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}, y)$ only activates the features whose weights are in $\boldsymbol{\theta}^{(y)}$. These features and weights are all we need to compute the joint log-probability $\log \mathrm{p}(\boldsymbol{x}, y)$ for each y. This is a key point: through this notation, we have converted the problem of computing the log-likelihood for a document-label pair $\langle \boldsymbol{x}^{(i)}, y^{(i)} \rangle$ into the computation of a vector inner product.

Estimation

The parameters of a multinomial distribution have a simple interpretation: they are the expected frequency for each word. Based on this interpretation, it is tempting to set the parameters empirically, as

$$\phi_{y,j} = \frac{\sum_{i:y^{(i)}=y} x_{i,j}}{\sum_{j'} \sum_{i:y^{(i)}=y} x_{i,j'}} = \frac{\text{count}(y, j)}{\sum_{j'} \text{count}(y, j')} \tag{1.47}$$

This is called a *relative frequency estimator*. It can be justified more rigorously as a *maximum likelihood estimate*.

Our prediction rule in Equation 1.38 is to choose $\hat{y}$ so as to maximize the joint probability $\mathrm{p}(\boldsymbol{x}, y)$. Maximum likelihood estimation proposes to choose the parameters $\boldsymbol{\phi}$ and $\boldsymbol{\mu}$ in much the same way. Specifically, we want to maximize the joint log-likelihood of some **training data**, which consists of a set of annotated examples where we observe both the text and the true label, $\{\boldsymbol{x}^{(i)}, y^{(i)}\}_{i \in 1 \ldots N}$. Based on the generative model that we have defined, the log-likelihood is:

$$L = \sum_i \log \mathrm{p}_{\text{mult}}(\boldsymbol{x}_i; \boldsymbol{\phi}_{y^{(i)}}) + \log \mathrm{p}_{\text{cat}}(y^{(i)}; \boldsymbol{\mu}). \tag{1.48}$$

Let's continue to focus on the parameters $\boldsymbol{\phi}$. Since $\mathrm{p}(y)$ is constant in L with respect to these parameters, we can forget it for now,

$$L(\boldsymbol{\phi}) = \sum_i \log \mathrm{p}_{\text{mult}}(\boldsymbol{x}^{(i)}; \boldsymbol{\phi}_{y^{(i)}}) \tag{1.49}$$

$$= \sum_i \log \frac{(\sum_j x_{i,j})!}{\prod_j x_{i,j}!} \prod_j \boldsymbol{\phi}_{y^{(i)},j}^{x_{i,j}} \tag{1.50}$$

$$= \sum_i \log \left[(\sum_j x_{i,j})! \right] - \sum_j \log (x_{i,j}!) + \sum_j x_{i,j} \log \boldsymbol{\phi}_{y^{(i)},j} \tag{1.51}$$

$$\propto \sum_j x_{i,j} \log \boldsymbol{\phi}_{y^{(i)},j}, \tag{1.52}$$

where I have abused notation by writing $\propto$ to indicate that the left side of Equation 1.52 is equal to the right side plus terms that are constant with respect to $\boldsymbol{\phi}$.

We would now like to optimize L, by taking derivatives with respect to $\boldsymbol{\phi}$. But before we can do that, we have to deal with a set of constraints:

$$\forall y, \sum_{j=1}^{V} \phi_{y,j} = 1 \tag{1.53}$$

We'll do this by adding a Lagrange multiplier. Solving separately for each label y, we obtain the resulting Lagrangian,

$$\ell[\boldsymbol{\phi}_y] = \sum_{i:Y^{(i)}=y} \sum_j x_{ij} \log \phi_{y,j} - \lambda(\sum_j \phi_{y,j} - 1) \tag{1.54}$$

We can now differentiate the Lagrangian with respect to the parameter of interest, setting $\frac{\partial \ell}{\partial \phi_{y,j}} = 0$,

$$0 = \sum_{i:Y^{(i)}=y} x_{i,j}/\phi_{y,j} - \lambda \tag{1.55}$$

$$\lambda \phi_{y,j} = \sum_{i:Y^{(i)}=y} x_{i,j} \tag{1.56}$$

$$\phi_{y,j} \propto \sum_{i:Y^{(i)}=y} x_{i,j} = \sum_i \delta(Y^{(i)} = y) x_{i,j}, \tag{1.57}$$

where I use two different notations for indicating the same thing: a sum over the word counts for all documents i such that the label $Y^{(i)} = y$. This gives a solution for each $\boldsymbol{\phi}_y$ up to a constant of proportionality. Now recall the constraint

$\forall y, \sum_{j=1}^{V} \boldsymbol{\phi}_{y,j} = 1$; this constraint arises because $\boldsymbol{\phi}_y$ represents a vector of probabilities for each word in the vocabulary. We can exploit this constraint to obtain an exact solution,

$$\phi_{y,j} = \frac{\sum_{i:Y^{(i)}=y} x_{i,j}}{\sum_{j'=1}^{V} \sum_{i:Y^{(i)}=y} x_{i,j'}} \tag{1.58}$$

$$= \frac{\text{count}(y, j)}{\sum_{j'=1}^{V} \text{count}(y, j')}. \tag{1.59}$$

This is exactly equal to the relative frequency estimator. A similar derivation gives $\mu_y \propto \sum_i \delta(Y^{(i)} = y)$, where $\delta(Y^{(i)} = y) = 1$ if $Y^{(i)} = y$ and 0 otherwise.

Smoothing and MAP estimation

If data is sparse, you may end up with values of $\phi = 0$. For example, the word *Bayesian* may have never appeared in a spam email yet, so the relative frequency estimate $\phi_{\text{SPAM},\textit{Bayesian}} = 0$. But choosing a value of 0 would allow this single feature to completely veto a label, since $\Pr(Y = \text{SPAM} \mid \boldsymbol{x}) = 0$ if $\boldsymbol{x}_{\textit{Bayesian}} > 0$.

This is undesirable, because it imposes high **variance**: depending on what data happens to be in the training set, we could get vastly different classification rules. One solution is to **smooth** the probabilities, by adding "pseudo-counts" of α to each count, and then normalizing.

$$\phi_{y,j} = \frac{\alpha + \sum_{i:Y^{(i)}=y} x_j^{(i)}}{\sum_{j'=1}^{V} \left(\alpha + \sum_{i:Y^{(i)}=y} x_{i,j'}\right)} = \frac{\alpha + \text{count}(y, j)}{V\alpha + \sum_{j'=1}^{V} \text{count}(y, j')} \tag{1.60}$$

This form of smoothing is called "Laplace smoothing", and it has a nice Bayesian justification, in which we extend the generative story to include $\boldsymbol{\phi}$ as a random variable (rather than as a parameter). The resulting estimate is called *maximum a posteriori*, or MAP.

Smoothing reduces **variance**, but it takes us away from the maximum likelihood estimate: it imposes a **bias**. In this case, the bias points towards uniform probabilities. Machine learning theory shows that errors on heldout data can be attributed to the sum of bias and variance. Techniques for reducing variance typically increase the bias, so there is a **bias-variance tradeoff**.[9]

[9] The bias-variance tradeoff is covered by Murphy (2012), but see Mohri et al. (2012) for a more formal treatment of this key concept in machine learning theory.

- Unbiased classifiers **overfit** the training data, yielding poor performance on unseen data.
- But if we set a very large smoothing value, we can **underfit** instead. In the limit of $\alpha \rightarrow \infty$, we have zero variance: it is the same classifier no matter what data we see! But the bias of such a classifier will be high.
- Navigating this tradeoff is hard. But in general, as you have more data, variance is less of a problem, so you just go for low bias.
- You may wonder if it is possible to choose a separate α_j for each word j, possibly to add larger amounts of smoothing to more common words. Indeed this is possible, and we will talk a great deal about more advanced smoothing techniques in Chapter 5. But I am unaware of any cases where this makes a major positive impact on classification.

Training, testing, and tuning (development) sets

We'll soon talk about more learning algorithms, but whichever one we apply, we will want to report its accuracy. Really, this is an educated guess about how well the algorithm will do on new data in the future.

To make an estimate of the accuracy, we need to hold out a separate "test set" from the data that we use for estimation (i.e., training, learning). Otherwise, if we measure accuracy on the same data that is used for estimation, we will badly overestimate the accuracy that we are likely to get on new data.

Recall that in addition to the parameters $\boldsymbol{\mu}$ and $\boldsymbol{\phi}$, which are learned on training data, we also have the amount of smoothing, α. This can be considered a "tuning" parameter, and it controls the tradeoff between overfitting and underfitting the training data. Where is the best position on this tradeoff curve? It's hard to tell in advance. Sometimes it is tempting to see which tuning parameter gives the best performance on the test set, and then report that performance. Resist this temptation! It will also lead to overestimating accuracy on truly unseen future data. For that reason, this is a sure way to get your research paper rejected; in a commercial setting, this mistake may cause you to promise much higher accuracy than you can deliver. Instead, you should split off a piece of your training data, called a "development set" (or "tuning set").

Sometimes, people average across multiple test sets and/or multiple development sets. One way to do this is to divide your data into "folds," and allow each fold to be the development set one time. This is called **K-fold cross-validation**. In the extreme, each fold is a single data point. This is called **leave-one-out**.

The Naïvety of Naïve Bayes

Naïve Bayes is simple to work with: estimation and prediction can done in closed form, and the nice probabilistic interpretation makes it relatively easy to extend the model in various ways. But Naïve Bayes makes assumptions which seriously limit its accuracy, especially in NLP.

- The multinomial distribution assumes that each word is generated independently of all the others (conditioned on the parameter ϕ_y). Formally, we assume conditional independence:

$$\mathrm{p}(\textit{naïve}, \textit{Bayes} \mid y) = \mathrm{p}(\textit{naïve} \mid y)\mathrm{p}(\textit{Bayes} \mid y). \tag{1.61}$$

- But this is clearly wrong, because words "travel together." To hone your intuitions about this, try and decide whether you believe

$$\mathrm{p}(\textit{naïve Bayes}) > \mathrm{p}(\textit{naïve})\mathrm{p}(\textit{Bayes}) \tag{1.62}$$

  or...

$$\mathrm{p}(\textit{naïve Bayes}) < \mathrm{p}(\textit{naïve})\mathrm{p}(\textit{Bayes}). \tag{1.63}$$

  Apply the chain rule!

Traffic lights Dan Klein makes this point with an example about traffic lights. In his hometown of Pittsburgh, there is a 1/7 chance that the lights will be broken, and both lights will be red. There is a 3/7 chance that the lights will work, and the north-south lights will be green; there is a 3/7 chance that the lights work and the east-west lights are green.

The *prior* probability that the lights are broken is 1/7. If they are broken, the conditional likelihood of each light being red is 1. The prior for them not being broken is 6/7. If they are not broken, the conditional likelihood of each individual light being red is 1/2.

Now, suppose you see that both lights are red. According to Naïve Bayes, the probability that the lights are broken is $1/7 \times 1 \times 1 = 1/7 = 4/28$. The probability that the lights are not broken is $6/7 \times 1/2 \times 1/2 = 6/28$. So according to naive Bayes, there is a 60% chance that the lights are not broken!

What went wrong? We have made an independence assumption to factor the probability $\mathrm{p}(R, R \mid \text{not-broken}) = \mathrm{p}_{\text{north-south}}(R \mid \text{not-broken})\mathrm{p}_{\text{east-west}}(R \mid \text{not-broken})$. But this independence assumption is clearly incorrect, because $\mathrm{p}(R, R \mid \text{not-broken}) = 0$.

Less Naïve Bayes? Of course we could decide not to make the naive Bayes assumption, and model $\mathrm{p}(R, R)$ explicitly. But this idea does not scale when the feature space is large — as it often is in NLP. The number of possible feature configurations grows exponentially, so our ability to estimate accurate parameters will suffer from high variance. With an infinite amount of data, we would be okay; but we never have that. Naïve Bayes accepts some bias, because of the incorrect modeling assumption, in exchange for lower variance.

Recap

- Instances can be represented as "bags of words", written as the vector $\boldsymbol{x}$.
- Feature functions combine the instance and its label into a single vector, $\boldsymbol{f}(\boldsymbol{x}, y)$.
- Classification can then be performed as a dot product $\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}, y)$.
- Naïve Bayes is a probabilistic classifier:
  - Define $\mathrm{p}(\boldsymbol{x}, \boldsymbol{y})$ via a *generative model*
  - Prediction: $\hat{y} = \operatorname{argmax}_y \mathrm{p}(\boldsymbol{x}^{(i)}, y)$
  - Learning:

$$\begin{aligned}
\boldsymbol{\theta} =& \operatorname*{argmax}_{\boldsymbol{\theta}} \mathrm{p}(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{\theta}) \\
\mathrm{p}(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{\theta}) =& \prod_i \mathrm{p}(\boldsymbol{x}^{(i)}, y^{(i)}; \boldsymbol{\theta}) = \prod_i \mathrm{p}(\boldsymbol{x}^{(i)} \mid y^{(i)}) \mathrm{p}(y^{(i)}) \\
\phi_{y,j} =& \frac{\sum_{i:Y^{(i)}=y} x_{ij}}{\sum_{i:Y^{(i)}=y} \sum_j x_{ij}} \\
\mu_y =& \frac{\text{count}(Y = y)}{N}
\end{aligned}$$

  This gives the maximum likelihood estimator (MLE; same as relative frequency estimator)

- The MLE is unbiased, but has high variance. We can navigate the bias-variance tradeoff by adding smoothing pseudo-counts α, reducing variance but adding bias.

Chapter 2

Discriminative learning

Naïve Bayes is a simple classifier, where both the prediction rule and the learning objective are based on the joint probability of labels and base features,

$$\log \mathrm{p}(y^{(i)}, \boldsymbol{x}^{(i)}) = \log \mathrm{p}(\boldsymbol{x}^{(i)} \mid y^{(i)}) + \log \mathrm{p}(y^{(i)}) \tag{2.1}$$

$$= \sum_j \log \mathrm{p}(x_{i,j} \mid y^{(i)}) + \log \mathrm{p}(y^{(i)}) \tag{2.2}$$

$$= \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) \tag{2.3}$$

Equation 2.2 shows the independence assumption that makes it possible to compute this joint probability: the probability of each base feature $x_{i,j}$ is mutually independent, after conditioning on the label $y^{(i)}$.

In the equations above, we define the **feature function** $\boldsymbol{f}(\boldsymbol{x}, y)$ so that it corresponds to "bag-of-words" features. These features do violate the assumption of conditional independence — for example, the probability of the word *naïve* is surely higher given the presence of the word *Bayes* — but the violation is relatively mild. However, to get really good performance on text classification and other language processing tasks, we will need to add many other types of features. Some of these features will capture parts of words, and others will capture multi-word units. For example:

- Prefixes, such as *anti-*, *im-*, and *un-*.
- Punctuation and capitalization.
- **Bigrams**, such as *not good*, *not bad*, *least terrible*, and higher-order **n-grams**.

These "rich" features tend to violate the Naïve Bayes independence assumption more severely. Consider what happens if we add feature capturing the word prefix. We then want to compute the probability,

$$\begin{aligned} Pr(\text{word} = \textit{impossible}, \text{prefix} = \textit{im-} \mid y) \approx & Pr(\text{prefix} = \textit{im-} \mid y) \\ & \times Pr(\text{word} = \textit{impossible} \mid y) \end{aligned} \tag{2.4}$$

To test the quality of the approximation, we can manipulate the original probability by applying the chain rule,

$$\begin{aligned} Pr(\text{word} = \textit{impossible}, \text{prefix} = \textit{im-} \mid y) = & Pr(\text{prefix} = \textit{im-} \mid \text{word} = \textit{impossible}, y) \\ & \times Pr(\text{word} = \textit{impossible} \mid y) \end{aligned} \tag{2.5}$$

But $Pr(\text{prefix} = \textit{im-} \mid \text{word} = \textit{impossible}, y) = 1$, since *im-* is guaranteed to be the prefix for the word *impossible*. Therefore,

$$Pr(\text{word} = \textit{impossible}, \text{prefix} = \textit{im-} \mid y) \tag{2.6}$$

$$= 1 \times Pr(\text{word} = \textit{impossible} \mid y)$$

$$\gg Pr(\text{prefix} = \textit{im-} \mid y) \times Pr(\text{word} = \textit{impossible} \mid y). \tag{2.7}$$

The final inequality is due to the fact that the probability of any given word starting with the prefix *im-* is much less than one, and it shows that Naïve Bayes will systematically underestimate the true probabilities of conjunctions of positively correlated features. To use such features, we will need learning algorithms that do not rely on an independence assumption.

2.1 Perceptron

In Naïve Bayes, the weights can be interpreted as parameters of a probabilistic model. But this model requires an independence assumption that usually does not hold, and limits our choice of features. Why not forget about probability and learn the weights in an error-driven way? The perceptron algorithm, shown in Algorithm 1, is one way to do this.[1]

What the algorithm says is this: if you make a mistake, increase the weights for features which are active with the correct label $y^{(i)}$, and decrease the weights

[1] The attentive reader will note that Algorithm 1 does not define the initial values of $\boldsymbol{\theta}$ or the index t. Initialization decisions are typically heuristic, and I prefer not to clutter the algorithm definition by committing to one initialization procedure or another. In this case, $\boldsymbol{\theta} = \mathbf{0}$ is a perfectly good choice. I have been similarly vague about the stopping criterion, but the text presents some alternatives. Counters like t should be assumed to begin at $t \leftarrow 1$ unless otherwise noted.

Algorithm 1 Perceptron learning algorithm

```
procedure PERCEPTRON($\boldsymbol{x}^{(1:N)}, y^{(1:N)}$)
    repeat
        Select an instance $i$
        $\hat{y} \leftarrow \text{argmax}_y \, \boldsymbol{\theta}_t^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y)$
        if $\hat{y} \neq y^{(i)}$ then
            $\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t + \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \boldsymbol{f}(\boldsymbol{x}^{(i)}, \hat{y})$
        else
            do nothing
    until tired
```

for features which are active with the guessed label $\hat{y}$. This is an **online learning** algorithm, since the classifier weights change after every example. This is different from Naïve Bayes, which computes corpus statistics and then sets the weights in a single operation — Naïve Bayes is a **batch learning** algorithm.[2]

The perceptron algorithm may seem like a cheap heuristic: Naïve Bayes has a solid foundation in probability, but now we are just adding and subtracting constants from the weights every time there is a mistake. Will this really work? In fact, there is some nice theory for the perceptron. To understand it, we must introduce the notion of **linear separability**:

Definition 1 (Linear separability)**.** *The dataset $\mathcal{D} = \{\langle \boldsymbol{x}^{(i)}, y^{(i)} \rangle\}_i$ is linearly separable iff there exists some weight vector $\boldsymbol{\theta}$ and some **margin** ρ such that for every instance $\langle \boldsymbol{x}^{(i)}, y^{(i)} \rangle$, the inner product of $\boldsymbol{\theta}$ and the feature function for the true label, $\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}, y^{(i)})$, is at least ρ greater than inner product of $\boldsymbol{\theta}$ and the feature function for every other possible label, $\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}, y')$.*

$$\exists \boldsymbol{\theta}, \rho > 0 : \forall \langle \boldsymbol{x}^{(i)}, y^{(i)} \rangle \in \mathcal{D}, \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) \geq \rho + \max_{y' \neq y^{(i)}} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y'). \tag{2.8}$$

Linear separability is important because of the following guarantee: if your data is linearly separable, then the perceptron algorithm will find a separator (Novikoff,

[2] Later in this chapter we will encounter a third class of learning algorithm, which is **iterative**. Such algorithms perform multiple updates to the weights (like perceptron), but are also **batch**, in that they have to use all the training data to compute the update.

1962).[3] So while the perceptron may seem heuristic, it is guaranteed to succeed — if the learning problem is easy enough.

How useful is this proof? Minsky and Papert (1969) note that the simple logical function of *exclusive-or* is not separable, and that a perceptron is therefore incapable of learning to mimic this function. But this is not just a problem for perceptron: any linear classification algorithm, including Naïve Bayes, will fail to learn this function. In natural language, we work in very high dimensional feature spaces, with thousands or millions of features. In these high-dimensional spaces, finding a separator becomes exponentially easier. Furthermore, later theoretical work showed that if the data is not separable, it is still possible to place an upper bound on the number of errors that the perceptron algorithm will make (Freund and Schapire, 1999).

Averaged perceptron

The perceptron iterates over the data repeatedly — until "tired", as described in Algorithm 1. If the data is linearly separable, it is guaranteed that the perceptron will eventually find a separator, and then we can stop. But if the data is not separable, the algorithm can *thrash* between two or more weight settings, never converging. In this case, how do we know that we can stop training, and how should we choose the final weights? An effective practical solution is to *average* the perceptron weights across all iterations.

This procedure is shown in Algorithm 2. The learning algorithm is nearly identical to the "vanilla" perceptron, but we also maintain a vector of the weight sums, $\boldsymbol{m}$. At the end of the learning procedure, we divide this sum by the total number of updates t, to compute the averaged weights, $\overline{\boldsymbol{\theta}}$. These averaged weights are then used to predict the labels of new data, such as examples in the test set. Even if the data is not separable, the averaged weights will eventually converge. One possible stopping criterion is to check the difference between the average weight vectors after each pass through the data: if the norm of the difference falls below some predefined threshold, we can stop iterating. Another stopping criterion is to hold out some data, and to measure the predictive accuracy on this heldout data (this is called a *development set* in chapter 1). When the accuracy on the heldout data starts to decrease, the learning algorithm has begun to **overfit**. At this point, it is probably best to stop; this stopping criterion is known as **early stopping**.

[3] It is also possible to prove an upper bound on the number of training iterations required to find the separator. Proofs like this are part of the field of **statistical learning theory**. Mohri et al. (2012) provide an excellent survey.

Algorithm 2 Averaged perceptron learning algorithm

```
procedure AVG-PERCEPTRON($\boldsymbol{x}^{(1:N)}, y^{(1:N)}$)
    repeat
        Select an instance $i$
        $\hat{y} \leftarrow \text{argmax}_y \, \boldsymbol{\theta}_t^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y)$
        if $\hat{y} \neq y^{(i)}$ then
            $\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t + \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \boldsymbol{f}(\boldsymbol{x}^{(i)}, \hat{y})$
            $\boldsymbol{m} \leftarrow \boldsymbol{m} + \boldsymbol{\theta}_{t+1}$
        else
            do nothing
    until tired
    $\overline{\boldsymbol{\theta}} \leftarrow \frac{1}{t}\boldsymbol{m}$
```

Generalization is the ability to make good predictions on instances that are not in the training data; it can be proved that averaging improves generalization, by computing an upper bound on the generalization error (Freund and Schapire, 1999; Collins, 2002).

2.2 Loss functions and large margin classification

Naïve Bayes chooses the weights $\boldsymbol{\theta}$ by maximizing the joint likelihood $\text{p}(\{\boldsymbol{x}^{(i)}, y^{(i)}\}_i)$. This is equivalent to maximizing the log-likelihood (due to the monotonicity of the log function), and also to **minimizing** the negative log-likelihood. This negative log-likelihood can therefore be viewed as a **loss function**,

$$\log \text{p}(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{\theta}) = \sum_{i=1}^{N} \log \text{p}(\boldsymbol{x}^{(i)}, y^{(i)}; \boldsymbol{\theta}) \tag{2.9}$$

$$\ell_{\text{NB}}(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) = -\log \text{p}(\boldsymbol{x}^{(i)}, y^{(i)}; \boldsymbol{\theta}) \tag{2.10}$$

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\text{argmin}} \sum_{i=1}^{N} \ell_{\text{NB}}(\boldsymbol{\theta}, \boldsymbol{x}^{(i)}, y^{(i)}) \tag{2.11}$$

This minimization problem is identical to the maximum-likelihood estimation problem that we solved in the previous chapter. Framing it as minimization may seem confusing and backwards, but loss functions provide a very general framework in which to compare many approaches to machine learning. For example, even though the perceptron is not a probabilistic model, it is also trying to minimize a **loss function**:

$$\ell_{\text{perceptron}}(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) = \begin{cases} 0, & y^{(i)} = \text{argmax}_y \, \boldsymbol{\theta}^\top \boldsymbol{f}(x_i, y) \\ 1, & \text{otherwise} \end{cases} \tag{2.12}$$

The perceptron loss — sometimes called the $0/1$ loss — has some pros and cons in comparison with the joint likelihood loss implied by Naive Bayes.

- ℓ_{NB} can suffer **infinite** loss on a single example, which suggests it will overemphasize some examples, and underemphasize others.
- $\ell_{\text{perceptron}}$ treats all errors equally. It only cares if the example is correct, and not about how confident the classifier was. Since we usually evaluate on accuracy or some related error-based metric, this is a better match.
- $\ell_{\text{perceptron}}$ is non-convex[4] and discontinuous. Although it is possible to bound the number of errors on the training data, finding the **global optimum** is intractable when the data is not separable.

We can fix this last problem by defining a loss function that behaves more nicely. To do this, let's define the **margin** as

$$\gamma(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) = \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \max_{y \neq y^{(i)}} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y) \tag{2.13}$$

The margin represents the separation between the score for the correct label $y^{(i)}$, and the score for the highest-scoring label. If the instance is classified incorrectly, the margin will be negative. The intuition behind "large-margin" learning algorithms is that it is not enough just to get the training data correct — we want the correct label to be separated from the other possible labels by a comfortable margin. We can use the margin to define a convex and continuous **hinge loss**,

$$\ell_{\text{hinge}}(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) = \begin{cases} 0, & \gamma(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) \geq 1, \\ 1 - \gamma(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}), & \text{otherwise} \end{cases} \tag{2.14}$$

Equivalently, we can write $\ell_{\text{hinge}}(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) = \left(1 - \gamma(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)})\right)_+$, where $(x)_+$ is equal to x if x is positive, and 0 otherwise. The hinge loss is zero if we have a

[4] A function f is convex iff $\alpha f(x_i) + (1-\alpha) f(x_j) \geq f(\alpha x_i + (1-\alpha) x_j)$, for all $\alpha \in [0, 1]$ and for all x_i and x_j on the domain of the function. Convexity implies that any local minimum is also a global minimum, and there are effective techniques for optimizing convex functions (Boyd and Vandenberghe, 2004).

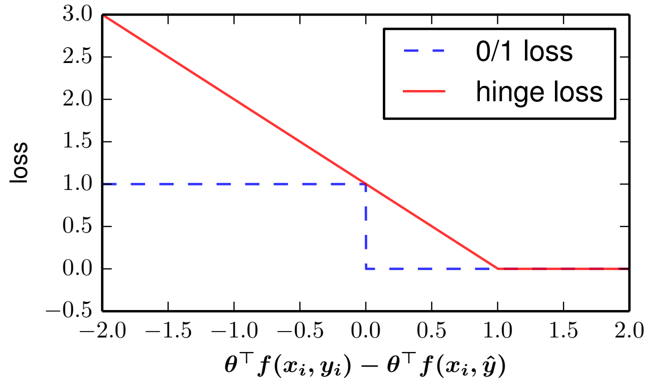


Figure 2.1: Hinge and perceptron loss functions

margin of at least 1 between the score for the true label and the best-scoring alternative, which we have written $\hat{y}$. The hinge and perceptron loss functions are shown in Figure 2.1. Note that the hinge loss is an upper bound on the perceptron loss.

Support vector machines

We can write the weight vector $\boldsymbol{\theta} = s\boldsymbol{u}$, where the **norm** of $\boldsymbol{u}$ is equal to one, $||\boldsymbol{u}||_2 = 1$.[5] Think of s as the magnitude and $\boldsymbol{u}$ as the direction of the vector $\boldsymbol{\theta}$. If the data is separable, there are many values of s that attain zero hinge loss. To see this, let us redefine the margin as,

$$\gamma(\boldsymbol{\theta}, \boldsymbol{x}^{(i)}, y^{(i)}) = \min_{y \neq y^{(i)}} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y) \tag{2.15}$$

$$= \min_{y \neq y^{(i)}} s(\boldsymbol{u}^\top \left(\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \boldsymbol{f}(\boldsymbol{x}^{(i)}, y)\right). \tag{2.16}$$

Based on this definition, if the unit vector $\boldsymbol{u}^*$ satisfies $\gamma(\boldsymbol{u}^*, \boldsymbol{x}^{(i)}, y^{(i)}) > 0$, then there is some smallest value s^* such that $\forall s \geq s^*, \gamma(s\boldsymbol{u}^*, \boldsymbol{x}^{(i)}, y^{(i)}) \geq 1$. This observation suggests that given many possible $\boldsymbol{\theta}$ that obtain zero hinge loss, we should choose the one with the smallest norm ($s = s^*$), since this entails making the least committment to the training data. This idea underlies the **Support Vector Machine** (SVM) classifier, which, in its most basic form, solves the following optimization

[5] The norm of a vector $||\boldsymbol{u}||_2$ is defined as, $||\boldsymbol{u}||_2 = \sqrt{\sum_j u_j^2}$.

problem,

$$\begin{aligned}
\min_{\boldsymbol{\theta}} . \quad & ||\boldsymbol{\theta}||_2^2 \\
s.t. \quad & \forall_i \ell_{\text{hinge}}(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) = 0.
\end{aligned} \tag{2.17}$$

Recall that $||\boldsymbol{\theta}||_2^2 = \sum_j \theta_j^2$.

In realistic settings, we do not know whether there is any feasible solution — that is, whether there exists any $\boldsymbol{\theta}$ so that the hinge loss on every training instance is zero. We therefore introduce a set of **slack variables** $\xi_i \geq 0$, which represent a sort of "fudge factor" for each instance i — instead of requiring that the hinge loss be exactly zero, we require that it be less than ξ_i. Ideally there would not be any slack, so we add the sum of the slack variables to the objective function to be minimized:

$$\begin{aligned}
\min_{\boldsymbol{\theta}} \quad & ||\boldsymbol{\theta}||_2^2 + C \sum_i \xi_i \\
s.t. \quad & \forall_i \ell_{\text{hinge}}(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) \leq \xi_i \\
& \forall_i \xi_i \geq 0.
\end{aligned} \tag{2.18}$$

Here C is a tunable parameter that controls the penalty on the slack variables. As $C \to \infty$, slack is infinitely expensive, and we can only find a solution if the data is separable. As $C \to 0$, slack becomes free, and there is a trivial solution at $\boldsymbol{\theta} = \mathbf{0}$, regardless of the data. Thus, C plays a similar role to the smoothing parameter in Naïve Bayes~(section 1.2), trading off between a close fit to the training data and better generalization. Like the smoothing parameter of Naïve Bayes, C must be set by the user, typically by maximizing performance on a heldout development set.

To solve the constrained optimization problem defined in Equation 2.18, we can use Lagrange multipliers to convert it into the unconstrained **primal form**,[6]

$$\min_{\boldsymbol{\theta}} \quad \frac{\lambda}{2} ||\boldsymbol{\theta}||_2^2 + \sum_i \ell_{\text{hinge}}(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}), \tag{2.19}$$

where λ is a tunable parameter that can be computed from the term C in Equation 2.18. A generic way to minimize such objective functions is **gradient descent**:

[6] An alternative **dual form** is used in the formulation of the kernel-based support vector machine, which supports non-linear classification. This is described briefly at the end of the chapter.

moving along the gradient (obtained by differentiating with respect to $\boldsymbol{\theta}$), until the gradient is equal to zero.[7]

Let us rewrite the primal form of the SVM optimization problem as follows:

$$L_{SVM} = \frac{\lambda}{2}||\boldsymbol{\theta}||_2^2 + \sum_i^N \ell_{\text{hinge}}(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) \tag{2.20}$$

$$= \frac{\lambda}{2}||\boldsymbol{\theta}||_2^2 + \sum_i^N (1 - \gamma(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}))_+ \tag{2.21}$$

$$= \frac{\lambda}{2}||\boldsymbol{\theta}||_2^2 + \sum_i^N (1 - \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) + \max_{y \neq y^{(i)}} \boldsymbol{f}(\boldsymbol{x}^{(i)}, y))_+ \tag{2.22}$$

Then the (sub)gradient of Equation 2.22 is:

$$\frac{\partial L_{SVM}}{\partial \boldsymbol{\theta}} = \lambda \boldsymbol{\theta} + \sum_i^N \delta_i (\max_{y \neq y^{(i)}} \boldsymbol{f}(\boldsymbol{x}^{(i)}, y) - \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)})) \tag{2.23}$$

$$\delta_i \triangleq \begin{cases} 1, & \gamma(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) < 1 \\ 0, & \text{otherwise.} \end{cases} \tag{2.24}$$

The term δ_i can be thought of as a gate: when the margin is less than one, the gradient includes the difference in features between the true label and the next best predicted label; when the margin is more than one, the gradient does not include this term. Because L_{SVM} is a convex function of $\boldsymbol{\theta}$, this gradient is equal to zero only at the global minimum. Gradient-based optimization techniques are discussed in section 2.4.

2.3 Logistic regression

Thus far, we have seen two broad classes of learning algorithms. Naïve Bayes is a probabilistic method, where learning is equivalent to estimating a joint probability distribution. Perceptron, support-vector machines (SVM), and passive-aggressive (PA) are all error-driven algorithms: the learning objective is to minimize the number of errors on the training data (perceptron), or to minimize a

[7] Because the hinge loss is not smooth, there is not a single gradient at the point at which the hinge loss is exactly equal to zero, but rather, a **subgradient set**. However, this is a theoretical issue that poses no difficulties in practice.

convex upper bound on the number of errors (SVM, PA). Both approaches have advantages: probability enables us to quantify uncertainty about the predicted labels, but error-driven learning typically leads to better performance on error-based performance metrics such as accuracy.

Logistic regression combines both of these advantages: it is error-driven like the perceptron and margin-based learning algorithms, but it is probabilistic like Naïve Bayes. To understand the motivation for logistic regression, first recall that Naïve Bayes selects weights to optimize the joint probability $\mathrm{p}(\boldsymbol{x}, y)$.

- We have used the chain rule to factor this joint probability as $\mathrm{p}(\boldsymbol{x}, y) = \mathrm{p}(\boldsymbol{x} \mid y) \times \mathrm{p}(y)$.
- But we could equivalently choose the alternative factorization $\mathrm{p}(\boldsymbol{x}, y) = \mathrm{p}(y \mid \boldsymbol{x}) \times \mathrm{p}(\boldsymbol{x})$.

In classification, we always know $\boldsymbol{x}$: these are the base features from which we predict y. So there is no need to model $\mathrm{p}(\boldsymbol{x})$; we really care only about the **conditional probability** $\mathrm{p}(y \mid \boldsymbol{x})$ — sometimes called the **likelihood**. Logistic regression defines this probability directly, in terms of the features $\boldsymbol{f}(\boldsymbol{x}, y)$ and the weights $\boldsymbol{\theta}$.

We can think of $\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}, y)$ as a scoring function for the compatibility of the base features $\boldsymbol{x}$ and the label y. This function is an unconstrained scalar; we would like to convert it to a probability. To do this, we first **exponentiate**, obtaining $\exp\left(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}, y)\right)$, which is guaranteed to be non-negative. Next, we need to **normalize**, dividing over all possible labels $y' \in \mathcal{Y}$. The resulting conditional probability is defined as,

$$\mathrm{p}(y \mid \boldsymbol{x}) = \frac{\exp\left(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}, y)\right)}{\sum_{y' \in \mathcal{Y}} \exp\left(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}, y')\right)}. \tag{2.25}$$

Given a dataset $\mathcal{D} = \{\langle \boldsymbol{x}^{(i)}, y^{(i)} \rangle\}_i$, the maximum-likelihood estimator for $\boldsymbol{\theta}$ is obtained by maximizing,

$$\begin{aligned}
L(\boldsymbol{\theta}) &= \log \mathrm{p}(\boldsymbol{y}^{(1:N)} \mid \boldsymbol{x}^{(1:N)}; \boldsymbol{\theta}) && (2.26)\\
&= \log \prod_i \mathrm{p}(y^{(i)} \mid \boldsymbol{x}^{(i)}; \boldsymbol{\theta}) && (2.27)\\
&= \sum_i \log \mathrm{p}(y^{(i)} \mid \boldsymbol{x}^{(i)}; \boldsymbol{\theta}) && (2.28)\\
&= \sum_i \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \log \sum_{y' \in \mathcal{Y}} \exp\left(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y')\right). && (2.29)
\end{aligned}$$

The final line is obtained by plugging in Equation 2.25 and taking the logarithm.[8,9] Inside the sum, we have the (additive inverse of the) **logistic loss**.

- In binary classification, we can write this as

$$\ell_{\text{logistic}}(\boldsymbol{\theta}; \boldsymbol{x}_i, y^{(i)}) = -(y^{(i)}\boldsymbol{\theta}^\top \boldsymbol{x}_i - \log\big(1 + \exp \boldsymbol{\theta}^\top \boldsymbol{x}_i\big)) \tag{2.30}$$

- In multi-class classification, we have,

$$\ell_{\text{logistic}}(\boldsymbol{\theta}; \boldsymbol{x}_i, y^{(i)}) = -(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \log \sum_{y' \in \mathcal{Y}} \exp \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y')) \tag{2.31}$$

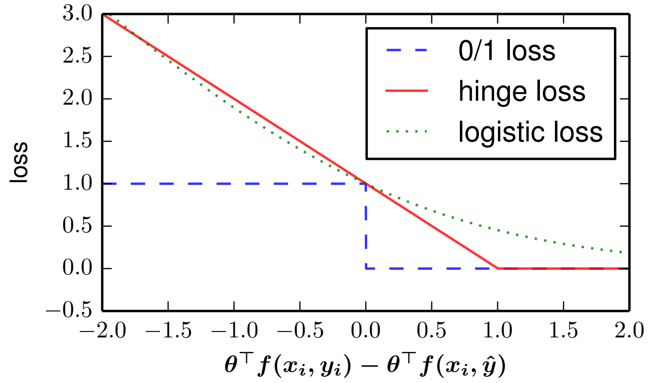


Figure 2.2: Hinge, perceptron, and logistic loss functions

The logistic loss is shown in Figure 2.2. Note that logistic loss is also an upper bound on the perceptron loss. A key difference from the perceptron and hinge losses is that logistic loss is *never* exactly zero: the objective function can always be improved by chosing the correct label with more confidence.

[8] Any reasonable base will work; if it is important to you to know which one to choose, then I suggest using base 2 if you are a computer scientist, and base e otherwise.

[9] The log-sum-exp term is very common in machine learning. It is numerically unstable because it will underflow if the inner product is small, and overflow if the inner product is large. Scientific computing libraries usually contain special functions for computing `logsumexp`, but with some thought, you should be able to see how to create an implementation that is numerically stable.

Regularization

As with the margin-based algorithms described in section 2.2, we can obtain better generalization by penalizing the norm of $\boldsymbol{\theta}$, by adding a term of $\frac{\lambda}{2}||\boldsymbol{\theta}||_2^2$ to the minimization objective. This is called L_2 regularization, because it includes the L_2 norm. It can be viewed as placing a zero-mean Gaussian prior distribution on each term of $\boldsymbol{\theta}$, because the log-likelihood under a zero-mean Gaussian is,

$$\log N(\theta_j; 0, \sigma^2) \propto -\frac{1}{2\sigma^2}\theta_j^2, \tag{2.32}$$

so that $\lambda = \frac{1}{\sigma^2}$.

The effect of this regularizer will cause the estimator to trade off conditional likelihood on the training data for a smaller norm of the weights, and this can help to prevent overfitting. Indeed, regularization is generally considered to be essential to estimating high-dimensional models, as we typically do in NLP. To see why, consider what would happen to the unregularized weight for a base feature j that was active in only one instance $\boldsymbol{x}^{(i)}$: the conditional likelihood could always be improved by increasing the weight for this feature, so that $\boldsymbol{\theta}_{(j,y^{(i)})} \to \infty$ and $\boldsymbol{\theta}_{(j,\tilde{y}\neq y^{(i)})} \to -\infty$, where (j, y) indicates the index of feature associated with $x_{i,j}$ and label y in $\boldsymbol{f}(\boldsymbol{x}^{(i)}, y)$.

Gradients

We will optimize $\boldsymbol{\theta}$ through gradient descent. Specific algorithms are described in section 2.4, but because the gradient of the logistic regression objective is illustrative, it is worth working out in detail. Let us begin with the logistic loss on a single example,

$$\ell(\boldsymbol{\theta}; \boldsymbol{x}_i, y^{(i)}) = -\left(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \log \sum_{y' \in \mathcal{Y}} \exp\left(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y')\right)\right) \tag{2.33}$$

$$\frac{\partial \ell}{\partial \boldsymbol{\theta}} = -\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) + \frac{1}{\sum_{y'' \in \mathcal{Y}} \exp\left(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y'')\right)} \times \sum_{y'} \exp\left(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y')\right) \times \boldsymbol{f}(\boldsymbol{x}^{(i)}, y') \tag{2.34}$$

$$= -\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) + \sum_{y'} \frac{\exp\left(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y')\right)}{\sum_{y'' \in \mathcal{Y}} \exp\left(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y'')\right)} \times \boldsymbol{f}(\boldsymbol{x}^{(i)}, y') \tag{2.35}$$

$$= -\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) + \sum_{y'} \mathrm{p}(y' \mid \boldsymbol{x}^{(i)}; \boldsymbol{\theta}) \times \boldsymbol{f}(\boldsymbol{x}^{(i)}, y') \tag{2.36}$$

$$= -\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) + E_{y|\boldsymbol{x}}[\boldsymbol{f}(\boldsymbol{x}^{(i)}, y)], \tag{2.37}$$

where the final step employs the definition of an expectation (section 1.1). The gradient thus has the pleasing interpretation as the difference between the observed feature counts $\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)})$ and the expected counts under the current model, $E_{y|\boldsymbol{x}}[\boldsymbol{f}(\boldsymbol{x}^{(i)}, y)]$. When these two count vectors are equal for a single example, there is nothing more to learn from this example; when they are equal in sum over the entire dataset, there is nothing more to learn from the dataset as a whole.

As we will see shortly, a simple online approach to gradient-based optimization is to take a step along the gradient. In (unregularized) logistic regression, this gradient-based optimization is a soft version of the perceptron. Put another way, in the case that $\text{p}(\boldsymbol{y} \mid \boldsymbol{x})$ is a delta function, $\text{p}(y \mid \boldsymbol{x}) = \delta(y = \hat{y})$, then the gradient step is exactly equal to the perceptron update.

If we add a regularizer $\frac{\lambda}{2}||\boldsymbol{\theta}||_2^2$, then this contributes $\lambda\boldsymbol{\theta}$ to the overall gradient:

$$L =\frac{\lambda}{2}||\boldsymbol{\theta}||_2^2 - \sum_{i=1}^{N} \left(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \log \sum_{y' \in \mathcal{Y}} \exp \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y') \right) \tag{2.38}$$

$$\frac{\partial L}{\partial \boldsymbol{\theta}} =\lambda\boldsymbol{\theta} - \sum_{i=1}^{N} \left(\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - E_{y|\boldsymbol{x}}[\boldsymbol{f}(\boldsymbol{x}^{(i)}, y)] \right) \tag{2.39}$$

2.4 Optimization

In Naïve Bayes, the gradient on the joint likelihood led us to a closed form solution for the parameters $\boldsymbol{\theta}$; in passive-aggressive, we obtained a solution for each individual update from a constrained optimization problem. In logistic regression and support vector machines (SVM), we have objective functions L.

- In logistic regression, L corresponds to the regularized negative log-likelihood,

$$L_{LR} = \frac{\lambda}{2}||\boldsymbol{\theta}||_2^2 - \sum_{i} \left(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \log \sum_{y} \exp \left(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y) \right) \right) \tag{2.40}$$

- In the support vector machine, L corresponds to the "primal form",

$$L_{SVM} = \frac{\lambda}{2}||\boldsymbol{\theta}||_2^2 - \sum_{i} (1 - \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) + \max_{y' \neq y^{(i)}} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y'))_+ \tag{2.41}$$

In both cases, the objective is convex, and there are many efficient algorithms for optimizing convex functions (Boyd and Vandenberghe, 2004). Most algorithms are based on the **gradient** $\frac{\partial L}{\partial \boldsymbol{\theta}}$, or on the subgradients, in the case of non-smooth objectives in which the gradient is not unique. This section will present the most frequently-used optimization algorithms, focusing on logistic regression. However, these algorithms can also be applied to the support vector machine objective with minimal modification.

Batch optimization

In batch optimization, all the data is kept in memory and iterated over many times. The logistic loss is smooth and convex, so we can find the global optimum using gradient descent,

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t - \eta_t \frac{\partial L}{\partial \boldsymbol{\theta}}, \tag{2.42}$$

where $\frac{\partial L}{\partial \boldsymbol{\theta}}$ is the gradient computed over the entire training set, and η_t is some **step size**. In practice, this can be very slow to converge, as the gradient can become infinitesimally small. Second-order (Newton) optimization obtains much better convergence rates by incorporating the inverse of the Hessian matrix,

$$H_{i,j} = \frac{\partial^2}{\partial w_i \partial w_j} L. \tag{2.43}$$

Unfortunately, in NLP problems, the Hessian matrix (which is quadratic in the number of parameters) is usually too big to deal with. A typical solution is to approximate the Hessian matrix via a **quasi-Newton optimization** technique, such as L-BFGS (Liu and Nocedal, 1989).[10] Quasi-Newton optimization packages are available in many scientific computing environments, and for most types of NLP practice and research, it is okay to treat them as black boxes. You will typically pass in a pointer to a function that computes the likelihood and gradient, and the solver will return a set of weights.

Online optimization

In online optimization, you consider one example (or a "mini-batch" of a few examples) at a time. **Stochastic gradient descent** (SGD) makes a stochastic online

[10] You can remember the order of the letters as "Large Big Friendly Giants." Does this help you?

approximation to the overall gradient. Here is the SGD update for logistic regression:

$$
\begin{aligned}
\boldsymbol{\theta}^{(t+1)} \leftarrow & \boldsymbol{\theta}^{(t)} - \eta_t \frac{\partial L_{LR}}{\partial \boldsymbol{\theta}} && (2.44)\\
= & \boldsymbol{\theta}^{(t)} - \eta_t \left(\lambda \boldsymbol{\theta}^{(t)} - \sum_{i}^{N} \left(\boldsymbol{f}(\boldsymbol{x}_i, y^{(i)}) - E_{y|\boldsymbol{x}}[\boldsymbol{f}(\boldsymbol{x}_i, y)] \right) \right) && (2.45)\\
= & (1 - \lambda \eta_t) \boldsymbol{\theta}^{(t)} + \eta_t \left(\sum_{i}^{N} \boldsymbol{f}(\boldsymbol{x}_i, y^{(i)}) - E_{y|\boldsymbol{x}}[\boldsymbol{f}(\boldsymbol{x}_i, y)] \right) && (2.46)\\
\approx & (1 - \lambda \eta_t) \boldsymbol{\theta}^{(t)} + N \eta_t \left(\boldsymbol{f}(\boldsymbol{x}_{i(t)}, y_{i(t)}) - E_{y|\boldsymbol{x}}[\boldsymbol{f}(\boldsymbol{x}_{i(t)}, y)] \right) && (2.47)
\end{aligned}
$$

where η_t is the **step size** at iteration t, and $\langle \boldsymbol{x}_{i(t)}, y_{i(t)} \rangle$ is an instance that is *randomly sampled* at iteration t. We can obtain a more compact form for SGD by folding the constant N into η_t and λ, so that $\tilde{\eta}_t = N \eta_t$ and $\tilde{\lambda} = \frac{\lambda}{N}$. This yields the form shown in Algorithm 3. A similar online algorithm can be derived for the SVM objective, using the subgradient in Equation 2.23.

Algorithm 3 Stochastic gradient descent for logistic regression

```
procedure SGD($\boldsymbol{x}^{(1:N)}, y^{(1:N)}, \eta, \lambda$)
    $t \leftarrow 1$
    repeat
        Select an instance $i$
        $\boldsymbol{\theta}^{(t+1)} \leftarrow (1 - \tilde{\lambda}\tilde{\eta}_t)\boldsymbol{\theta}^{(t)} + \tilde{\eta}_t \left( \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - E_{y|\boldsymbol{x}}[\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)})] \right)$
        $t \leftarrow t + 1$
    until tired
```

As above, the expectation is equal to a weighted sum over the labels,

$$
E_{y|\boldsymbol{x}}[\boldsymbol{f}(\boldsymbol{x}^{(i)}, y)] = \sum_{y' \in \mathcal{Y}} \mathrm{p}(y' \mid \boldsymbol{x}^{(i)}; \boldsymbol{\theta}) \boldsymbol{f}(\boldsymbol{x}^{(i)}, y'). \tag{2.48}
$$

Again, note how similar this update is to the perceptron.

The theoretical foundation for SGD assumes that each training instance is randomly sampled (thus the name "stochastic"), but in practice, it is typical to stream through the data sequentially. It is often useful to select not a single instance, but a **mini-batch** of K instances. In this case, we would scale η_t and λ by $\frac{N}{K}$. The gradients over mini-batches will be lower variance approximations of the true gradient, and it is possible to parallelize the computation of the gradient for each instance in the mini-batch.

A key question for SGD is how to set the learning rates η_t. It can be proven that SGD will converge if $\eta_t = \eta_0 t^{-\alpha}$ for $\alpha \in [1, 2]$; however, convergence may be very slow. In practice, η_t may also be fixed to a small constant, like 10^-3. In either case, it is typical to try a set of different values, and see which minimizes the objective L most quickly. For more on stochastic gradient descent, as applied to a number of different learning algorithms, see (Zhang, 2004) and (Bottou, 1998). Murphy (2012) traces SGD to Nemirovski and Yudin (1978).

AdaGrad

There are a number of ways to improve on stochastic gradient descent (Bottou et al., 2016). For NLP applications, a popular choice is use an **adaptive** step size, which can be different for every feature (Duchi et al., 2011). Features that occur frequently are likely to be updated frequently, so it is best to use a small step size; rare features will be updated infrequently, so it is better to take larger steps. The **AdaGrad** (adaptive gradient) algorithm achieves this behavior by storing the sum of the squares of the gradients for each feature, and rescaling the learning rate by its inverse:

$$\begin{aligned}\boldsymbol{g}_t =& \lambda\boldsymbol{\theta} - \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) + \sum_{y' \in \mathcal{Y}} \mathrm{p}(y' \mid \boldsymbol{x}^{(i)})\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) && (2.49)\\ \theta_j^{(t+1)} \leftarrow& \theta_j^{(t)} - \frac{\eta}{\sqrt{\sum_{t'=1}^{t} g_{t,j}^2}} g_{t,j}, && (2.50)\end{aligned}$$

where j iterates over features in $\boldsymbol{f}(\boldsymbol{x}, y)$. AdaGrad seems to require less careful tuning of η, and Dyer (2014) reports that $\eta = 1$ works for a wide range of problems.

2.5 Additional topics in classification*

Passive-aggressive

In online learning, rather than seeking the feasible $\boldsymbol{\theta}$ with the smallest norm, we might instead prefer to make the smallest magnitude **change** to $\boldsymbol{\theta}$, while meeting the hinge loss constraint for instance $\langle \boldsymbol{x}^{(i)}, y^{(i)} \rangle$. Specifically, at each step t, we solve the following optimization problem:

$$\begin{aligned} \min w. \quad & \frac{1}{2}||\boldsymbol{\theta} - \boldsymbol{\theta}_t||^2 + C\xi_t \\ s.t. \quad & \ell_{\text{hinge}}(\boldsymbol{\theta}; \boldsymbol{x}_i, y^{(i)}) \leq \xi_t, \xi_t \geq 0 \end{aligned} \tag{2.51}$$

By forming another Lagrangian, it is possible to show that the solution to Equation 2.51 is,

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \tau_t(\boldsymbol{f}(y^{(i)}, \boldsymbol{x}^{(i)}) - \boldsymbol{f}(\hat{y}, \boldsymbol{x}^{(i)})) \tag{2.52}$$

$$\tau_t = \min\left(C, \frac{\ell(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)})}{||\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \boldsymbol{f}(\boldsymbol{x}^{(i)}, \hat{y})||^2}\right), \tag{2.53}$$

This algorithm is called **Passive-Aggressive** (PA; Crammer et al., 2006), because it is passive when the margin constraint is satisfied, but it aggressively changes the weights to satisfy the constraints if necessary.[11] PA is error-driven like the perceptron, and the update is nearly identical: the only difference is the learning rate τ_t, which depends on the amount of loss incurred by instance i, the norm of the difference in feature vectors between the predicted and correct labels, and the hyperparameter C, which places an upper bound on the step size. As with the perceptron, it is possible to apply weight averaging to PA, which can improve generalization. PA allows more explicit control than the Averaged Perceptron, due to the C parameter: when C is small, we make very conservative adjustments to $\boldsymbol{\theta}$ from each instance, because the slack variables aren't very expensive; when C is large, we make large adjustments to avoid using the slack variables.

Other regularizers

In Equation 2.38, we proposed to **regularize** the logistic regression estimator by penalizing the squared L_2 norm, $||\boldsymbol{\theta}||_2^2$. However, this is not the only way to penalize large weights; we might prefer some other norm, such as $L_0 = ||\boldsymbol{\theta}||_0 = \sum_j \delta(\theta_j \neq 0)$, which applies a constant penalty for each non-zero weight. This norm can be thought of as a form of **feature selection**: optimizing the L_0-regularized conditional likelihood is equivalent to trading off the log-likelihood against the number of active features. Reducing the number of active features is desirable because the resulting model will be fast, low-memory, and should generalize well, since features that are not very helpful will be pruned away. Unfortunately, the L_0

[11] A related algorithm without slack variables is called MIRA, for Margin-Infused Relaxed Algorithm (Crammer and Singer, 2003).

norm is non-convex and non-differentiable; optimization under L_0 regularization is NP-hard, meaning that it can be solved efficiently only if P=NP (Ge et al., 2011).

A useful alternative is the L_1 norm, which is equal to the sum of the absolute values of the weights, $||\boldsymbol{\theta}||_1 = \sum_j |\theta_j|$. The L_1 norm is convex, and can be used as an approximation to L_0 (Tibshirani, 1996). Moreover, the L_1 norm also performs feature selection, by driving many of the coefficients to zero; it is therefore known as a **sparsity inducing regularizer**. Gao et al. (2007) compare L_1 and L_2 regularization on a suite of NLP problems, finding that L_1 regularization generally gives similar test set accuracy to L_2 regularization, but that L_1 regularization produces models that are between ten and fifty times smaller, because more than 90% of the feature weights are set to zero.

The L_1 norm does not have a gradient at $\theta_j = 0$, so we must instead optimize the L_1-regularized objective using **subgradient** methods. The associated stochastic subgradient descent algorithms are only somewhat more complex than conventional SGD; Sra et al. (2012) survey approaches for estimation under L_1 and other regularizers.

Other views of logistic regression

Logistic regression is so named because in the binary case where $y \in \{0, 1\}$, we are performing a regression of $\boldsymbol{x}$ against $\boldsymbol{y}$, after passing the inner product $\boldsymbol{\theta}^\top \boldsymbol{x}$ through a logistic transformation to obtain a probability. However, it goes by many other names:

- Logistic regression is also called **maximum conditional likelihood** (MCL), because it is based on maximizing the conditional likelihood $\text{p}(y \mid \boldsymbol{x})$.
- Logistic regression can be viewed as part of a larger family of **generalized linear models** (GLMs), which include other "link functions," such as the probit function. If you use the R software environment, you may be familiar with `glmnet`, a widely-used package for estimating GLMs.
- In the neural networks literature, the multivariate analogue of the logistic transformation is sometimes called a **softmax** layer, because it "softly" identifies the label y that maximizes the activation function $\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}, y)$.

In the early NLP literature, logistic regression is frequently called **maximum entropy** (Berger et al., 1996). This is due to an alternative formulation, which tries to find the maximum entropy probability function that satisfies moment-matching constraints. The moment matching constraints specify that the empirical counts of each label-feature pair should match the expected counts:

$$\forall j, \sum_{i=1}^{N} f_j(\boldsymbol{x}^{(i)}, y^{(i)}) = \sum_{i=1}^{N} \sum_{y \in \mathcal{Y}} \mathrm{p}(y \mid \boldsymbol{x}^{(i)}; \boldsymbol{\theta}) f_j(\boldsymbol{x}^{(i)}, y) \tag{2.54}$$

Note that this constraint will be met exactly when the derivative of the likelihood function (Equation 2.37) is equal to zero. However, this constraint can be met for many values of $\boldsymbol{\theta}$, so which should we choose?

The **entropy** of the conditional likelihood $\mathrm{p}_{y|\boldsymbol{x}}$ is,

$$H(\mathrm{p}_{y|\boldsymbol{x}}) = -\sum_{\boldsymbol{x} \in \mathcal{X}} \mathrm{p}_{\boldsymbol{x}}(\boldsymbol{x}) \sum_{y \in \mathcal{Y}} \mathrm{p}_{y|\boldsymbol{x}}(y \mid \boldsymbol{x}) \log \mathrm{p}_{y|\boldsymbol{x}}(y \mid \boldsymbol{x}), \tag{2.55}$$

where $\mathrm{p}_{\boldsymbol{x}}(\boldsymbol{x})$ is the probability of observing the base features $\boldsymbol{x}$. We compute an empirical estimate of the entropy by summing over all the instances in the training set,

$$\tilde{H}(\mathrm{p}_{y|\boldsymbol{x}}) = -\frac{1}{N} \sum_{i} \sum_{y \in \mathcal{Y}} \mathrm{p}_{y|\boldsymbol{x}}(y \mid \boldsymbol{x}^{(i)}) \log \mathrm{p}_{y|\boldsymbol{x}}(y \mid \boldsymbol{x}^{(i)}). \tag{2.56}$$

If the entropy is large, the likelihood function is smooth across possible values of y; if it is small, the likelihood function is sharply peaked at some preferred value; in the limiting case, the entropy is zero if $\mathrm{p}(y \mid x) = 1$ for some y. By saying we want a maximum-entropy classifier, we are saying we want to make the weakest commitments possible, while satisfying the moment-matching constraints from Equation 2.54. The solution to this constrained optimization problem is identical to the maximum conditional likelihood (logistic-loss) formulation we considered in the previous section. This view of logistic regression is arguably a little dated, but it is useful to understand, especially when reading classic papers from the 1990s. For a tutorial on maximum entropy, see `http://www.cs.cmu.edu/afs/cs/user/aberger/www/html/tutorial/tutorial.html`.

2.6 Summary of learning algorithms

Having seen several learning algorithms, it is natural to ask which is best in various situations.

Naïve Bayes *Pros*: easy to implement; estimation is very fast, requiring only a single pass over the data; assigns probabilities to predicted labels; controls overfitting with smoothing parameter. *Cons*: the joint likelihood is arguably the wrong objective to optimize; often has poor accuracy, especially with correlated features.

Perceptron and PA *Pros*: easy to implement; online learning means it is not necessary to store all data in memory; error-driven learning means that accuracy is typically high, especially after averaging. *Cons*: not probabilistic, which can be bad in pipeline architectures, when the output of one system becomes the input for another; non-averaged perceptron performs poorly if data is not separable; hard to know when to stop learning.

Support vector machine *Pros*: optimizes an error-based metric, usually resulting in high accuracy; overfitting is controlled by a regularization parameter. *Cons*: batch learning requires black-box optimization; not probabilistic.

Logistic regression *Pros*: error-driven and probabilistic; overfitting is controlled by a regularization parameter. *Cons*: batch learning requires black-box optimization; logistic loss sometimes gives lower accuracy than hinge loss, due to overtraining on correctly-labeled examples.

Table 2.1 summarizes some properties of Naïve Bayes, perceptron, PA, and logistic regression. SVM is left out because it is identical to PA on most of these dimensions, except for the estimation procedure, which typically employs a black-box convex optimization package. In non-probabilistic settings, I usually reach for averaged perceptron first if I am coding from scratch, and SVM if I am using a library of learning algorithms such as `sklearn`. If probabilities are necessary, I use logistic regression.

What about non-linear classification?

The feature spaces that we consider in NLP are usually huge, so non-linear classification can be quite difficult. When the feature dimension V is larger than the number of instances N — often the case in NLP — you can always learn a linear classifier that will perfectly classify your training instances.[12] This makes selecting an appropriate **non-linear** classifier especially difficult. Nonetheless, there are some approaches to non-linear learning in NLP:

[12] Assuming your feature matrix is full-rank.

- The most common approach is to define $\boldsymbol{f}(\boldsymbol{x}, y)$ to contain conjunctions or other nonlinear combinations of the base features in $\boldsymbol{x}$. For example, a bigram feature such as ⟨*coffee house*⟩ will not fire unless both base features ⟨*coffee*⟩ and ⟨*house*⟩ also fire. More generally, we can define non-linear transformations such as the element-wise product $\boldsymbol{x} \circ \boldsymbol{x}$ and the cross-product $\boldsymbol{x} \otimes \boldsymbol{x}$.
- **Kernel**-based learning is based on similarity between instances; it can be seen as a generalization of k**-nearest-neighbors**, which classifies instances by considering the label of the k most similar instances in the training set (Hastie et al., 2009). The resulting decision boundary will be non-linear in general. Kernel functions can be designed to compute the similarity between structured objects, such as strings, bags-of-words, sequences, trees, and general graphs. Such methods will be discussed briefly in chapter 18.
- Boosting (Freund et al., 1999) and decision tree algorithms (Schmid, 1994) learn non-linear conjunctions of features. These methods sometimes do well on NLP tasks, but are used less frequently in contemporary research, especially as the field increasingly emphasizes big data and simple classifiers.
- More recent work has shown how **deep learning** can perform non-linear classification, by passing the inputs through a series of non-linear transformations. These methods will be reviewed in chapter 21; surveys are offered by Goldberg (2015) and Cho (2015).

	Naive Bayes	Logistic Regression	Perceptron	PA
Objective	Joint likelihood	Conditional likelihood	0/1 loss	Hinge loss
	$\max \sum_i \log \mathrm{p}(\boldsymbol{x}^{(i)}, y^{(i)})$	$\max \sum_i \log \mathrm{p}(y^{(i)} \mid \boldsymbol{x}^{(i)})$	$\min \sum_i \delta(y^{(i)}, \hat{y})$	$\sum_i [1 - \gamma(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)})]_+$
estimation	$\theta_{ij} = \frac{c(x_i, y=j)+\alpha}{c(y=j)+V\alpha}$	$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}} = \sum_i \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - E[\boldsymbol{f}(\boldsymbol{x}^{(i)}, y)]$	$\boldsymbol{\theta}^{(t)} \leftarrow \boldsymbol{\theta}^{(t-1)} + \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \boldsymbol{f}(\boldsymbol{x}^{(i)}, \hat{y})$	$\boldsymbol{\theta}^{(t)} \leftarrow \boldsymbol{\theta}^{(t-1)} + \tau_t(\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \boldsymbol{f}(\boldsymbol{x}^{(i)}, \hat{y}))$
tuning	smoothing α	regularizer $\lambda \|\|\boldsymbol{\theta}\|\|_2^2$	weight averaging	slack penalty C
complexity	$\mathcal{O}(NV)$	$\mathcal{O}(NVT)$	$\mathcal{O}(NVT)$	$\mathcal{O}(NVT)$
easy?	very	not really	yes	yes
probabilities?	yes	yes	no	no
features?	no	yes	yes	yes

Table 2.1: Comparison of classifiers. N = number of examples, V = number of features, T = number of instances.

(leave this page blank or the next page gets messed up)

Chapter 3

Linguistic applications of classification

Having learned some techniques for classification, let's now see how they can be applied to typical problems in natural language technology.

3.1 Sentiment and opinion analysis

A popular NLP technology is automatically determining the "sentiment" or "opinion polarity" of documents such as product reviews and social media posts. For example, marketers are interested to know how people respond to advertisements, services, and products (Hu and Liu, 2004); social scientists are interested in how emotions are affected by phenomena such as the weather (Hannak et al., 2012), and how both opinions and emotions spread over social networks (Coviello et al., 2014; Miller et al., 2011). In the field of **digital humanities**, literary scholars track plot structures through the flow of sentiment across a novel (Jockers, 2015). A comprehensive analysis of this broad literature is beyond the scope of this chapter, but see survey manuscripts by Pang and Lee (2008) and Liu (2015).

Sentiment analysis can be framed as a fairly direct application of document classification, assuming reliable labels can be obtained. In the simplest case, sentiment analysis can be treated as a two or three-class problem, with sentiments of POSITIVE, NEGATIVE, and possibly NEUTRAL. Such annotations could be annotated by hand, or obtained automatically through a variety of means:

- Tweets containing happy emoticons can be marked as positive, sad emoticons as negative (Read, 2005; Pak and Paroubek, 2010).

- Reviews with four or more stars can be marked as positive, two or fewer stars as negative (Pang et al., 2002).
- Statements from politicians who are voting **for** a given bill are marked as positive (towards that bill); statements from politicians voting against the bill are marked as negative (Thomas et al., 2006).

After obtaining the annotations, several design decisions may be taken in construction of the feature vector $\boldsymbol{f}(\boldsymbol{x}, y)$:

Preprocessing One question is whether the vocabulary should be case sensitive: do we distinguish *great*, *Great*, and GREAT? What about *coooooool*? In social media text, this sort of **expressive lengthening** can cause the vocabulary size to explode (Brody and Diakopoulos, 2011); we might want to somehow **normalize** the text (Sproat et al., 2001) to collapse the vocabulary again.

A related issue is that suffixes may be irrelevant to the sentiment orientation of each word: for example, *love*, *loved*, and *loving* are all positive, so perhaps we should eliminate the suffix and group them together. The removal of these suffixes is called **stemming** when it is done at the character level (leaving roots like *lov-*), and is called **lemmatization** when the goal is to identify the underlying base word (in this case, *love*). Both of these methods will be discussed in detail in chapter 9 and chapter 8.

Still another preprocessing decision involves **tokenization**: breaking the text into tokens. This is more complicated than simply looking for whitespace, since we may want to tokenize items such as *well-bred* into $\langle$*well*, *bred*$\rangle$, *isn't* into $\langle$*is*, *n't*$\rangle$; at the same time, we would like to keep *U.S.* as a single token. This too will be discussed in chapter 8.

Vocabulary In some cases, it is preferable not to include all words in the vocabulary. Words such as *the*, *to*, and *and* seem intuitively to play little role in expressing sentiment or opinion, yet they are very frequent; removing these **stopwords** may therefore improve the classifier. This is typically done by creating a list and simply matching all items on the list. More aggressively, we might assume that sentiment is typically carried by **adjectives** and **adverbs** (see Chapter **??**), and therefore we could focus on these words (Hatzivassiloglou and McKeown, 1997; Turney, 2002). However, Pang et al. (2002) find that in their case, eliminating non-adjectives causes the performance of the classifier to decrease.

Count or binary? Finally, we may consider whether we want our feature vector to include the **count** of each word, or its mere **presence**. This gets at a subtle limitation of linear classification: two *failure*s may be worse than one, but is it really twice as bad? A more flexible classifier could assign diminishing weight to each additional instance, but this is hard to do in the linear classification framework, and its hard to see how much the weight should diminish. Pang et al. (2002) take a simpler approach, using binary presence/absence indicators in the feature vector: $f_i(\boldsymbol{x}, y) \in \{0, 1\}, \forall i$. They find that classifiers trained on these binary feature vectors outperform classifiers trained on count-based features.

A more challenging version of opinion analysis is to determine not just the class of a review, but its rating on a numerical scale (Pang and Lee, 2005). If the scale is continuous, we might take a regression approach, identifying a set of weights $\boldsymbol{\theta}$ so as to minimize the squared error of a predictor $\hat{y} = \boldsymbol{\theta}^\top \boldsymbol{x} + b$, where b is an offset. We can remove the offset by adding a feature to $\boldsymbol{x}$ whose value is always 1; the corresponding weight in $\boldsymbol{\theta}$ is then equivalent to b. Least squares regularization has a closed form solution,

$$\boldsymbol{\theta} =(\mathbf{X}^\top \mathbf{X})^{-1}\mathbf{X}^\top \boldsymbol{y}, \tag{3.1}$$

where $\boldsymbol{y}$ is a column vector of size N, containing all ratings in the training data, and $\mathbf{X}$ is an $N \times D$ matrix containing all D features for all N instances. If we place an L2 regularizer on $\boldsymbol{\theta}$, with penalty $\lambda||\boldsymbol{\theta}||_2^2$, the resulting problem is called **ridge regression**. It too has a closed form solution,

$$\boldsymbol{\theta} =(\mathbf{X}^\top \mathbf{X} + \lambda \mathbb{I})^{-1}\mathbf{X}^\top \boldsymbol{y}. \tag{3.2}$$

If the rating scale is discrete, $y \in \{1, 2, \ldots, K\}$, we can take a **ranking** approach (Crammer and Singer, 2001), in which scores $\boldsymbol{\theta}^\top \boldsymbol{x}$ are discretized into ranks, by also learning a set of boundaries, $b_0 = -\infty \leq b_1 \leq \ldots \leq b_K$. The learning algorithm consists in making perceptron-like updates to both $\boldsymbol{\theta}$ and $\boldsymbol{b}$. This approach is ideal for settings like predicting a 1-10 rating or a grade (A - F); instead of learning one vector $\boldsymbol{\theta}$ for every rank, we can learn a single $\boldsymbol{\theta}$, and then just partition the output space.

[todo: Other topics to cover:]

- subjectivity
- sentence-level versus document-level sentiment
- negation and the role of syntax

- targeted sentiment
- Stance classification

3.2 Word sense disambiguation

Consider the the following headlines:

(3.1) *Iraqi head seeks arms*

(3.2) *Prostitutes appeal to Pope*

(3.3) *Drunk gets nine years in violin case*[1]

They are ambiguous because they contain words that have multiple meanings, or **senses**. Word Sense Disambiguation (WSD) is the problem of identifying the intended sense of each word token in a document. WSD is part of a larger field of research called **lexical semantics**, which is concerned with meanings of the words.

Problem definition

Part-of-speech ambiguity (e.g., noun versus verb, as in *she is* ***heading*** *out of town*) is usually considered to be a different problem from WSD. Here we are focusing on ambiguity between senses that are all the same part-of-speech, and in part-of-speech tagging evaluations, it is often assumed that the correct part-of-speech has already been identified. [todo: why?] From a linguistic perspective, senses are not really properties of words, but of **lemmas**, which are groups of inflected forms, e.g. $(arm/N, arms/N)$, $(arm/V, arms/V, armed/V, arming/V)$, where arm/N indicates the word *arm* tagged as a noun (V is for verb). So the WSD problem can be defined as identifying the correct sense for each word token from an inventory associated for the word's lemma.

How many word senses?

Words (lemmas) may have *many* more than two senses. For example, the word *serve* would seem to have at least the following senses:

- [FUNCTION]: *The tree stump served as a table*

[1] These examples, and many more, can be found at `http://www.ling.upenn.edu/~beatrice/humor/headlines.html`

- [ENABLE]: *His evasive replies only served to heighten suspicion*
- [DISH]: *We serve only the rawest fish here*
- [ENLIST]: *She served her country in the marines*
- [JAIL]: *He served six years in Alcatraz*
- [TENNIS]: *Nobody can return his double-reverse spin serve*
- [LEGAL]: *They were served with subpoenas*[2]

How do we know that these senses are really different? Linguists often design tests for this purpose, and one such test is to construct a **zeugma**, which combines antagonistic senses in an uncomfortable way:

(3.4) *Which flight serves breakfast?*

(3.5) *Which flights serve Tuscon?*

(3.6) **Which flights serve breakfast and Tuscon?*[3]

The asterisk is a linguistic notation for utterances which would not be judged to be grammatical by fluent speakers of a language. To the extent that you think that (3.6) is ungrammatical, you should agree that (3.4) and (3.5) refer to distinct senses of the lemma *serve*.

The WSD task: Output What should the output of WSD be? What are the possible senses for each word? We could just look in the dictionary. But rather than using a traditional dictionary, WSD research is dominated by a computational resource called WORDNET (`http://wordnet.princeton.edu`). WordNet is organized in terms of lemmas rather than words. An example of a wordnet entry is shown in Figure 3.1

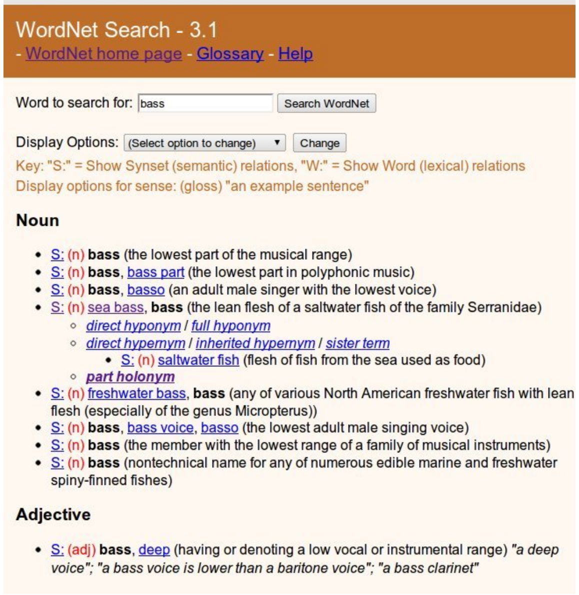
WordNet consists of roughly 100,000 **synsets**, groups of words or phrases with an identical meaning. (e.g., $\{$CHUMP^1, FOOL^2, SUCKER^1, MARK$^9\}$). A lemma is **polysemous** if it participates in multiple synsets. Besides **synonymy**, WordNet also describes many other lexical relationships, including:

antonymy x means the opposite of y, e.g. FRIEND-ENEMY;

hyponymy x is a special case of y, e.g. RED-COLOR; the inverse relationship is **hypernymy**;

[2] Examples from Dan Klein's lecture notes, `http://www.cs.berkeley.edu/~klein/cs294-7/SP07%20cs294%20lecture%205%20--%20maximum%20entropy%20(6pp).pdf`

[3] I believe this example is from Jurafsky and Martin (2009) [todo: but check].



WordNet Search - 3.1
- WordNet home page - Glossary - Help

Word to search for: bass Search WordNet

Display Options: (Select option to change) ▾ Change
Key: "S:" = Show Synset (semantic) relations, "W:" = Show Word (lexical) relations
Display options for sense: (gloss) "an example sentence"

Noun

- S: (n) **bass** (the lowest part of the musical range)
- S: (n) **bass**, bass part (the lowest part in polyphonic music)
- S: (n) **bass**, basso (an adult male singer with the lowest voice)
- S: (n) sea bass, **bass** (the lean flesh of a saltwater fish of the family Serranidae)
  - *direct hyponym* / *full hyponym*
  - *direct hypernym* / *inherited hypernym* / *sister term*
    - S: (n) saltwater fish (flesh of fish from the sea used as food)
  - ***part holonym***
- S: (n) freshwater bass, **bass** (any of various North American freshwater fish with lean flesh (especially of the genus Micropterus))
- S: (n) **bass**, bass voice, basso (the lowest adult male singing voice)
- S: (n) **bass** (the member with the lowest range of a family of musical instruments)
- S: (n) **bass** (nontechnical name for any of numerous edible marine and freshwater spiny-finned fishes)

Adjective

- S: (adj) **bass**, deep (having or denoting a low vocal or instrumental range) *"a deep voice"; "a bass voice is lower than a baritone voice"; "a bass clarinet"*

Figure 3.1: Example wordnet entry, from `http://wordnet.princeton.edu`

meronymy x is a part of y, e.g., WHEEL-BICYCLE; the inverse relationship is **holonymy**.

WordNet has played a big role in helping WSD move from toy systems to to large-scale quantitative evaluations. However, some have argued that WordNet's sense granularity is too fine (Ide and Wilks, 2006); more fundamentally, the premise that word senses can be differentiated in a task-neutral way has been criticized as linguistically naïve (Kilgarriff, 1997). One way of testing this question is to ask whether people tend to agree on the appropriate sense for example sentences: according to Mihalcea et al. (2004), humans agree on roughly 70% of examples using WordNet senses; far better than chance, but perhaps less than we might like.

A range of tasks have been proposed for WSD:

- **Synthetic** data: different words are conflated (*banana-phone*), the system must identify the original word.
- **Lexical sample**: disambiguate a few target words (e.g., *plant* etc). This is what was used in the first large-scale WSD evaluation, SENSEVAL-1 (1998).[todo: citation]
- **All-words** WSD: a sense must be identified for every token.

- A **semantic concordance** is a corpus in which each open-class word (nouns, verbs, adjectives, and adverbs) is tagged with its word sense from the target dictionary or thesaurus.
- SEMCOR is a semantic concordance built from 234K tokens of the Brown corpus.
  As of $\textit{Sunday}_n^1$ $\textit{night}_n^1$ *there* $\textit{was}_v^4$ *no* $\textit{word}_n^2 \ldots$

WSD as Classification

So, how can we tell living *plants* from manufacturing *plants*? The key information often lies in the **context**:

(3.7) *Town officials are hoping to attract new manufacturing plants through weakened environmental regulations.*

(3.8) *The endangered plant plays an important role in the local ecosystem.*

Bag-of-words models are a very typical approach. For example,

$$\begin{gathered}f(y, \textit{bank}, \textit{I went to the bank to deposit my paycheck}) = \\ \{\langle \textit{went}, y\rangle : 1, \langle \textit{deposit}, y\rangle : 1, \langle \textit{paycheck}, y\rangle : 1\}\end{gathered}$$

Some examples:[4]

- *bank*[FINANCIAL]:

  a an and are ATM Bonnie card charges check Clyde criminals deposit famous for get I much My new overdraft really robbers the they think to too two went were

- *bank*[RIVER]:

  a an and big campus cant catfish East got grandfather great has his I in is Minnesota Mississippi muddy My of on planted pole pretty right River The the there University walk Wets

An extension of bag-of-words models is to encode the position of each context word, e.g.,

$$\begin{aligned}&f(y, \textit{bank}, \textit{I went to the bank to deposit my paycheck}) = \\ &\{\langle i-3, \textit{went}, y\rangle : 1, \langle i+2, \textit{deposit}, y\rangle : 1, \langle i+4, \textit{paycheck}, y\rangle : 1\}\end{aligned}$$

[4] todo: reconcile with examples above

Jurafsky and Martin (2009) call these **collocation features**. Other approaches include more information about the sentence structure, such as the part-of-speech tag for each word, and the words with which it is syntactically linked in the sentence (see chapter 12).

After deciding on the features, we can train a classifier to predict the right sense of each word — assuming enough labeled examples can be accumulated. This is difficult, because each polysemous lemma requires its own training set: having a good classifier for *bank* is of no help at all towards disambiguating *plant*. For this reason, **unsupervised** and **semisupervised** methods are particularly popular for WSD (Yarowsky, 1995). We will talk about related methods in chapter 4 and chapter 20. Unsupervised methods typically lean heavily on the heuristic "one sense per discourse", meaning roughly that a lemma will have a consistent sense throughout any given document. Based on this heuristic, we can propagate information from high-confidence instances to lower-confidence instances in the same document. For a survey on word sense disambiguation, see Navigli (2009).

3.3 Other applications

- Author identification
- Author demographics, maybe
- Language classification

3.4 Evaluating text classification

In any text classification setting, it is critical to reserve a held-out test set, and use this data for only one purpose: to evaluate the overall accuracy of a single classifier. Using this data more than once would cause your estimated accuracy to be overly optimistic. Since it is typically necessary to set hyperparameters or perform feature selection, you may need to construct various "tuning" or "development" sets, but these should not intersect with the test data. For more details, see section 1.2.

There are a number of ways to evaluate classifier performance. The simplest is **accuracy**: the number of correct predictions, divided by the total number of instances.

Why isn't this always the right choice? Suppose we were building a classifier to detect whether an essay receives a passing grade. Due perhaps to grade inflation, 95% of all essays receive a passing grade. This means that a classifier that always says "pass" will get 95% accuracy. But this classifier isn't telling us anything useful at all.

Another way to evaluate this classifier is in terms of its **precision** and **recall**. For each label $y \in \mathcal{Y}$, we define a **positive** instance as one that the classifier labels as $Y_i = y$, and a **negative** instance as one that the classifier labels as $Y_i \neq y$. We can then define four quantities:

True positive positive and correct, TP

False positive positive but incorrect, FP

True negative negative and correct, TN

False negative negative and incorrect, FN.

From these quantities, we can then define the **recall** and **precision**:

$$r = \frac{TP}{TP + FN} \tag{3.3}$$

$$p = \frac{TP}{TP + FP} \tag{3.4}$$

The recall is the proportion of positive labels among those that **should** have been labeled as positive (for some label y). The precision is the proportion of positive labels among those that **were** labeled as positive. Our "always pass" classifier above would have 100% recall for the positive label, but 95% precision. It would have 0% recall for the negative label, and undefined precision.

The **f-measure** is the harmonic mean of recall and precision,

$$F = \frac{2 \times r \times p}{r + p}. \tag{3.5}$$

F-measure is a classic measure of classifier performance for binary classification problems with unbalanced class distribution. Sometimes it is called $F1$, as there are generalizations of f-measure in which the precision is multiplied by some constant β^2.

Macro-F1 is the average f-measure across several classes. In a multi-class problem with unbalanced class distributions, the macro-F1 is a balanced measure of how well the classifier recognizes each class. In **micro-F1**, we compute true positives, false positives, and false negatives for each class, and then add them up before computing a single f-measure. This metric is balanced across instances rather than classes, so will weight each class in proportion to how frequently it appears.

[todo: ROC curves and AUC]

Chapter 4

Learning without supervision

So far we've assumed the following setup:

- A **training set** where you get observations $\boldsymbol{x}_i$ and labels y_i
- A **test set** where you only get observations $\boldsymbol{x}_i$

What if you never get labels y_i? For example, suppose you are trying to do word sense disambiguation. You get a bunch of text, and you suspect that there are at least two different meanings for the word *concern*. But you don't have any labels for specific instances in which this word is used. What can you?

As described in chapter 3, in supervised word sense disambiguation, we often build feature vectors from the words that appear in the context of the word that we are trying to disambiguate. For example, for the word *concern*, the immediate context might typically include words from one of the following two groups:

1. *services, produces, banking, pharmaceutical, energy, electronics*
2. *about, said, that, over, in, with, had*

Now suppose we were to scatterplot each instance of *concern* on a graph, so that the x-axis is the density of words in group 1, and the y-axis is the density of words in group 2. In such a graph, shown in Figure 4.1, two or more blobs might emerge. These blobs would correspond to the different sense of *concern*.

But in reality, we don't know the word groupings in advance.[1] We have to try to apply the same idea in a very high dimensional space, where every word gets its own dimension — and most dimensions are irrelevant!

[1] One approach, which we do not consider here, would be to get them from some existing resource, such as the dictionary definition (Lesk, 1986).

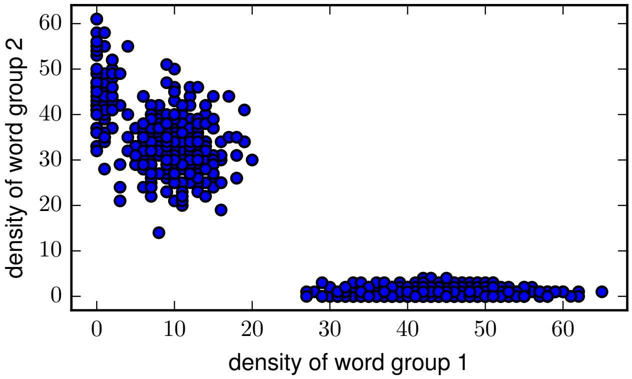


Figure 4.1: Counts of words from two different context groups

Now here's a related scenario, from a different problem. Suppose you download thousands of news articles, and make a scatterplot, where each point corresponds to a document: the x-axis is the frequency of the word *hurricane*, and the y-axis is the frequency of the word *election*. Again, three clumps might emerge: one for documents that are largely about the hurricane, another for documents largely about the election, and a third clump for documents about neither topic.

These examples are intended to show that we can find structure in data, even without labels — just look for clumps in the scatterplot of features. But again, in reality we cannot make scatterplots of just two words; we may have to consider hundreds or thousands of words. It would be impossible to visualize such a high-dimensional scatterplot, so we will need to design algorithmic approaches to finding these groups.

4.1 K-means clustering

You might know about classic clustering algorithms like K**-means**. These algorithms maintain a cluster assignment for each instance, and a central location for each cluster. They them repeatedly update the cluster assignments and the locations, until convergence. Pseudocode for K-means is shown in Algorithm 4.

K-means can used to find coherent clusters of documents in high-dimensional data. When we assign each point to its nearest center, we are choosing which cluster it is in; when we re-estimate the location of the centers, we are determining the defining characteristic of each cluster. K-means is a classic algorithmic that has

Algorithm 4 K-means clustering algorithm

```
procedure $K$-MEANS($\boldsymbol{x}_{1:N}$)
    Initialize cluster centers $\mu_k \leftarrow$ Random()
    repeat
        for all $i$ do
            Assign each point to the nearest cluster: $z_i \leftarrow \min_k \text{Distance}(\boldsymbol{x}_i, \mu_k)$
        for all $k$ do
            Recompute each cluster center from the points in the cluster: $\mu_k \leftarrow \frac{1}{\sum_i \delta(z_i = k)} \sum_i \delta(z_i = k) \boldsymbol{x}_i$
    until converged
```

been used and modified in thousands of papers (Jain, 2010); for an application of K-means to word sense induction, see Pantel and Lin (2002).

Of the many variants of K-means, one that is particularly relevant for our purposes is called **soft K-means**. The key difference is that instead of directly assigning each point $\boldsymbol{x}_i$ to a specific cluster z_i, soft K-means assigns each point a **distribution** over clusters $q_i(z_i)$, so that $\sum_k q_i(k) = 1$, and $\forall_k 0 \leq q_i(k) \leq 1$. The centroid of each cluster is then computed from a **weighted average** of the points in the cluster, where the weights are taken from the q distribution.

We will now explore a more principled, statistical version of soft K-means, called **expectation-maximization** (EM) clustering. By understanding the statistical principles underlying the algorithm, we can extend it in a number of ways.

4.2 The Expectation-Maximization (EM) Algorithm

Let's go back to the Naïve Bayes model:

$$\log \mathrm{p}(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{\phi}, \mu) = \sum_i \log \mathrm{p}(\boldsymbol{x}_i \mid y_i; \boldsymbol{\phi}) \mathrm{p}(y_i; \mu) \tag{4.1}$$

For example, $\boldsymbol{x}$ can describe the documents that we see today, and $\boldsymbol{y}$ can correspond to their labels. But suppose we never observe y_i? Can we still do anything with this model?

Since we don't know $\boldsymbol{y}$, let's marginalize it:

$$\log \mathrm{p}(\boldsymbol{x}) = \sum_{i}^{N} \log \mathrm{p}(\boldsymbol{x}_i) \tag{4.2}$$

$$= \sum_i \log \sum_{y_i} \mathrm{p}(\boldsymbol{x}_i \mid y_i; \boldsymbol{\phi}) \mathrm{p}(y_i; \mu) \tag{4.3}$$

$$\tag{4.4}$$

We will estimate the parameters $\boldsymbol{\phi}$ and μ by maximizing the log-likelihood of $\boldsymbol{x}_{1:N}$, which is our (unlabeled) observed data. Why is this a good thing to maximize? If we don't have labels, discriminative learning is impossible (there's nothing to discriminate), so maximum likelihood is all we have.

Unfortunately, maximizing $\log P(\boldsymbol{x})$ directly is intractable. So to estimate this model, we must employ approximation. We do this by introducing an **auxiliary variable $\boldsymbol{q}_i$**, for each y_i. We want $\boldsymbol{q}_i$ to be a **distribution**, so we have the usual constraints: $\sum_y q_i(y) = 1$ and $\forall y, q_i(y) \geq 0$. In other words, q_i defines a probability distribution over $\mathcal{Y}$, for each instance i.

Now since $\frac{q_i(y)}{q_i(y)} = 1$, we can multiply the right side by this ratio and preserve the equality,

$$\log \mathrm{p}(\boldsymbol{x}) = \sum_i \log \sum_{y_i} \mathrm{p}(\boldsymbol{x}_i \mid y_i; \boldsymbol{\phi}) \mathrm{p}(y_i; \mu) \frac{q_i(y)}{q_i(y)} \tag{4.5}$$

$$= \sum_i \log E_q \left[\frac{\mathrm{p}(\boldsymbol{x}_i \mid y; \boldsymbol{\phi}) \mathrm{p}(y; \mu)}{q_i(y)} \right], \tag{4.6}$$

by the definition of expectation, $E_q [f(x)] = \sum_x q(x) f(x)$. Note that $E_q [\cdot]$ just means the expectation under the distribution q.

Now we apply **Jensen's inequality**, which says that because $\log$ is a concave function, we can push it inside the expectation, and obtain a lower bound.

$$\log \mathrm{p}(\boldsymbol{x}) \geq \sum_i E_q \left[\log \frac{\mathrm{p}(\boldsymbol{x}_i \mid y; \boldsymbol{\phi}) \mathrm{p}(y_i; \mu)}{q_i(y)} \right] \tag{4.7}$$

$$\mathcal{J} = \sum_i E_q [\log \mathrm{p}(\boldsymbol{x}_i \mid y; \boldsymbol{\phi})] + E_q [\log \mathrm{p}(y; \mu)] - E_q [\log q_i(y)] \tag{4.8}$$

By maximizing $\mathcal{J}$, we are maximizing a lower bound on the joint log-likelihood $\log \mathrm{p}(\boldsymbol{x})$. Now, $\mathcal{J}$ is a function of two sets of arguments:

- the distributions q_i for each i
- the parameters μ and $\boldsymbol{\phi}$

We'll optimize with respect to each of these in turn, holding the other one fixed.

Step 1: the E-step

First, we expand the expectation in the lower bound as:

$$\mathcal{J} = \sum_i E_q \left[\log \mathrm{p}(\boldsymbol{x}_i \mid y; \boldsymbol{\phi})\right] + E_q \left[\log \mathrm{p}(y; \mu)\right] - E_q \left[\log q_i(y)\right] \tag{4.9}$$

$$= \sum_i \sum_y q_i(y) \left(\log \mathrm{p}(\boldsymbol{x}_i \mid y; \boldsymbol{\phi}) + \log \mathrm{p}(y; \mu) - \log q_i(y)\right) \tag{4.10}$$

As in Naïve Bayes, we have a "sum-to-one" constraint: in this case, $\sum_y q_i(y) = 1$. Once again, we incorporate this constraint into a Lagrangian:

$$\mathcal{J}_q = \sum_i^N \sum_{y \in \mathcal{Y}} q_i(y) \left(\log \mathrm{p}(\boldsymbol{x}_i \mid y; \boldsymbol{\phi}) + \log \mathrm{p}(y; \mu) - \log q_i(y)\right) + \lambda_i (1 - \sum_y q_i(y)) \tag{4.11}$$

We then optimize by taking the derivative and setting it equal to zero:

$$\frac{\partial \mathcal{J}_q}{\partial q_i(y)} = \log \mathrm{p}(\boldsymbol{x}_i \mid y; \boldsymbol{\phi}) + \log \mathrm{p}(y; \boldsymbol{\theta}) - \log q_i(y) - 1 - \lambda_i \tag{4.12}$$

$$\log q_i(y) = \log \mathrm{p}(\boldsymbol{x}_i \mid y; \boldsymbol{\phi}) + \log \mathrm{p}(y; \mu) - 1 - \lambda_i \tag{4.13}$$

$$q_i(y) \propto \mathrm{p}(\boldsymbol{x}_i \mid y; \boldsymbol{\phi}) \mathrm{p}(y; \mu) = \mathrm{p}(\boldsymbol{x}_i, y; \boldsymbol{\phi}, \mu) \tag{4.14}$$

Since q_i is defined over the labels $\mathcal{Y}$, we normalize it as,

$$q_i(y) = \frac{\mathrm{p}(\boldsymbol{x}_i, y; \boldsymbol{\phi}, \mu)}{\sum_{y' \in \mathcal{Y}} \mathrm{p}(\boldsymbol{x}_i, y'; \boldsymbol{\phi}, \mu)} = \mathrm{p}(y \mid \boldsymbol{x}_i; \boldsymbol{\phi}, \mu) \tag{4.15}$$

After normalizing, each $q_i(y)$ — which is the soft distribution over clusters for data $\boldsymbol{x}_i$ — is set to the posterior probability $\mathrm{p}(y \mid \boldsymbol{x}_i)$ under the current parameters $\mu, \boldsymbol{\phi}$. This is called the E-step, or "expectation step," because it is derived from updating the bound on the expected likelihood under $q(\boldsymbol{y})$. Note that although we introduced the Lagrange multipliers λ_i as additional parameters, we were able to drop these parameters because we solved for $q_i(y)$ to a constant of proportionality.

Step 2: the M-step

Next, we hold $q(\boldsymbol{y})$ fixed and maximize the bound with respect to the parameters, $\boldsymbol{\phi}$ and μ. Lets focus on $\boldsymbol{\phi}$, which parametrizes the likelihood, $\mathrm{p}(\boldsymbol{x} \mid y; \boldsymbol{\phi})$. Again, we have a constraint that $\sum_j^V \phi_{y,j} = 1$, so we start by forming a Lagrangian,

$$\mathcal{J}_\phi = \sum_i^N \sum_{y \in \mathcal{Y}} q_i(y) \left(\log \mathrm{p}(\boldsymbol{x}_i \mid y; \boldsymbol{\phi}) + \log \mathrm{p}(y; \mu) - \log q_i(y)\right) + \sum_{y \in \mathcal{Y}} \lambda_y (1 - \sum_j^V \phi_{y,j}). \tag{4.16}$$

Again, we solve by setting the derivative equal to zero:

$$\frac{\partial \mathcal{J}_\phi}{\partial \phi_{y,j}} = \sum_i^N q_i(y) \frac{x_{i,j}}{\phi_{y,j}} - \lambda_y \tag{4.17}$$

$$\lambda_h \phi_{y,j} = \sum_i^N q_i(y) x_{i,j} \tag{4.18}$$

$$\phi_{y,j} \propto \sum_i^N q_i(y) x_{i,j}. \tag{4.19}$$

Now because $\sum_j^V \phi_{y,j} = 1$, we can normalize as follows,

$$\phi_{y,j} = \frac{\sum_i^N q_i(y) x_{i,j}}{\sum_{j' < V} \sum_i^N q_i(y) x_{i,j'}} \tag{4.20}$$

$$= \frac{E_q\left[\text{count}(y, j)\right]}{E_q\left[\text{count}(y)\right]}, \tag{4.21}$$

where $j \in \{1, 2, \ldots, V\}$ indexes base features, such as words.

So $\boldsymbol{\phi}_y$ is now equal to the relative frequency estimate of the **expected counts** under the distribution $q(y)$.

- As in supervised Naïve Bayes, we can apply smoothing to add α to all these counts.
- The update for μ is identical: $\mu_y \propto \sum_i q_i(y)$, the expected proportion of cluster $Y = y$. If needed, we can add smoothing here too.
- So, everything in the M-step is just like Naïve Bayes, except that we use expected counts rather than observed counts.

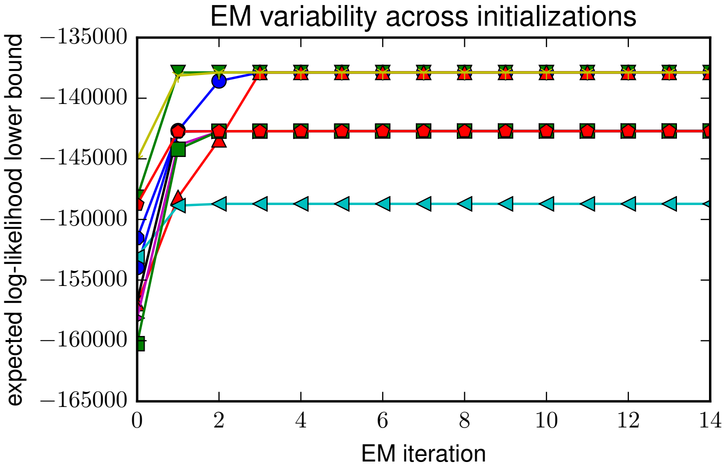


Figure 4.2: Sensitivity of expectation maximization to initialization

This is the M-step for a model in which the likelihood $P(\boldsymbol{x} \mid \boldsymbol{y})$ is multinomial. For other likelihoods, there my be no closed-form solution for the parameters in the M-step. We may therefore run gradient-based optimization at each M-step, or we may simply take a single step along the gradient step and then return to the E-step (Berg-Kirkpatrick et al., 2010).

Coordinate ascent

Algorithms that alternate between updating various subsets of the parameters are called "coordinate ascent" algorithms.

The objective function $\mathcal{J}$ is **biconvex**, meaning that it is separately convex in $q(\boldsymbol{y})$ and $\langle \mu, \boldsymbol{\phi} \rangle$, but it is not jointly convex in all terms. In the coordinate ascent algorithm that we have defined, each step is guaranteed not to decrease $\mathcal{J}$. This is sometimes called "hill climbing", because you never go down. Specifically, EM is guaranteed to converge to a **local optima** — a point which is as good or better than any of its immediate neighbors. But there may be many such points, and the overall procedure is **not** guaranteed to find a global maximum. Figure 4.2 shows the objective function for EM with ten different random initializations: while the objective function increases monotonically in each run, it converges to several different values.

The fact that there is no guarantee of global optimality means that initialization is important: where you start can determine where you finish. This is not true in the supervised learning algorithms that we have considered, such as logistic regression — although deep learning algorithms do suffer from this problem. But for logistic regression, and for many other supervised learning algorithms, we don't need to worry about initialization, because it won't affect our ultimate solution: we are guaranteed to reach the global minimum. Recent work on **spectral learning** has sought to obtain similar guarantees for "latent variable" models, such as the case we are considering now, where $\boldsymbol{x}$ is observed and y is latent. This work is briefly touched on in section 4.4.

Variants In **hard EM**, each q_i distribution assigns probability of 1 to a single $\hat{y}_i$, and probability of 0 to all others (Neal and Hinton, 1998). This is similar in spirit to K-means clustering. In problems where the space $\mathcal{Y}$ is large, it may be easier to find the maximum likelihood value $\hat{y}$ than it is to compute the entire distribution $q_i(y)$. Spitkovsky et al. (2010) show that hard EM can outperform standard EM in some cases.

Another variant of the coordinate ascent procedure combines EM with stochastic gradient descent (SGD). In this case, we can do a local E-step at each instance i, and then immediately make an gradient update to the parameters $\langle \mu, \boldsymbol{\phi} \rangle$. This is particularly relevant in cases where there is no closed form solution for the parameters, so that gradient ascent will be necessary in any case. This algorithm is called "incremental EM" by Neal and Hinton (1998), and online EM by Sato and Ishii (2000) and Cappé and Moulines (2009). Liang and Klein (2009) apply a range of different online EM variants to NLP problems, obtaining better results than standard EM in many cases.

How many clusters?

All along, we have assumed that the number of clusters $K = \#|\mathcal{Y}|$ is given. In some cases, this assumption is valid. For example, the dictionary or WordNet might tell us the number of senses for a word. In other cases, the number of clusters should be a tunable parameter: some readers may want a coarse-grained clustering of news stories into three or four clusters, while others may want a fine-grained clusterings into twenty or more. But in many cases, we will have choose K ourselves, with little outside guidance.

One solution is to choose the number of clusters to maximize some computable quantity of the clustering. First, note that the likelihood of the training data will always increase with K. For example, if a good solution is available for $K = 2$, then we can always obtain that same solution at $K > 2$; usually we can find an even better solution by fitting the data more closely. The Akaike Information Crition (AIC; Akaike, 1974) solves this problem by minimizing a linear combination of the log-likelihood and the number of model parameters, AIC $= 2m - 2\mathcal{L}$, where m is the number of parameters and $\mathcal{L}$ is the log-likelihood. Since the number of parameters increases with the number of clusters K, the AIC may prefer more parsimonious models, even if they do not fit the data quite as well.

Another choice is to maximize the **predictive likelihood** on heldout data $\boldsymbol{x}^{(h)}_{1:N_h}$. This data is not used to estimate the model parameters $\boldsymbol{\phi}$ and μ; we can compute the predictive likelihood on this data by keeping the parameters $\boldsymbol{\phi}$ and μ fixed, and running a single iteration of the E-step. In document clustering or **topic modeling** (Blei, 2012), a typical approach is to split each instance (document) in half. We use the first half to estimate $q_i(z_i)$, and then on the second half we compute the expected log-likelihood,

$$\ell_i = \sum_z q_i(z) \left(\log \mathrm{p}(\boldsymbol{x}_i \mid z; \boldsymbol{\phi}) + \log \mathrm{p}(z; \mu)\right). \tag{4.22}$$

On heldout data, this quantity will not necessarily increase with the number of clusters K, because for high enough K, we are likely to overfit the training data. Thus, choosing K to maximize the predictive likelihood on heldout data will limit the extent of overfitting. Note that in general we cannot analytically find the K that maximizes either AIC or the predictive likelihood, so we must resort to grid search: trying a range of possible values of K, and choosing the best one.

Finally, it is worth mentioning an alternative approach, called **Bayesian nonparametrics**, in which the number of clusters K is treated as another latent variable. This enables statistical inference over a set of models with a variable number of clusters; this is not possible with EM, but there are several alternative inference procedures that are suitable for this case (Murphy, 2012), including MCMC (section 4.4). Reisinger and Mooney (2010) provide a nice example of Bayesian nonparametrics in NLP, applying it to unsupervised word sense induction.

4.3 Applications of EM

EM is not really an "algorithm" like, say, quicksort. Rather, it is a framework for learning with missing data. The recipe for using EM on a problem of interest is:

- Introduce latent variables $\boldsymbol{z}$, such that it is easy to write the probability $P(\mathcal{D}, \boldsymbol{z})$, where $\mathcal{D}$ is your observed data; it should also be easy to estimate the associated parameters, given knowledge of $\boldsymbol{z}$.

- Derive the E-step updates for $q(\boldsymbol{z})$, which is typically factored as $q(\boldsymbol{z}) = \prod_i q_{z_i}(z_i)$, where i is an index over instances.
- The M-step updates typically correspond to the soft version of some supervised learning algorithm, like Naïve Bayes.

Some more applications of this basic setup are presented here.

Word sense clustering

In the "demos" folder, you can find a demonstration of expectation-maximization for word sense clustering. I assume we know that there are two senses, and that the senses can be distinguished by the contextual information in the document. The basic framework is identical to the clustering model of EM as presented above.

Semi-supervised learning

Nigam et al. (2000) offer another application of EM: **semi-supervised learning**. They apply this idea to document classification in the classic "20 Newsgroup" dataset, in which each document is a post from one of twenty newsgroups from the early days of the internet.

In the setting considered by Nigam et al. (2000), we have labels for some of the instances, $\langle \boldsymbol{x}^{(\ell)}, \boldsymbol{y}^{(\ell)} \rangle$, but not for others, $\langle \boldsymbol{x}^{(u)} \rangle$. The question they pose is: can unlabeled data improve learning? If so, then we might be able to get good performance from a smaller number of labeled instances, simply by incorporating a large number of unlabeled instances. This idea is called **semi-supervised learning**, because we are learning from a combination of labeled and unlabeled data; the setting is described in much more detail in chapter 20.

As in Naïve Bayes, the learning objective is to maximize the joint likelihood,

$$\log \mathrm{p}(\boldsymbol{x}^{(\ell)}, \boldsymbol{x}^{(u)}, \boldsymbol{y}^{(\ell)}) = \log \mathrm{p}(\boldsymbol{x}^{(\ell)}, \boldsymbol{y}^{(\ell)}) + \log \mathrm{p}(\boldsymbol{x}^{(u)}) \tag{4.23}$$

We treat the labels of the unlabeled documents as missing data — in other words, as a latent variable. In the E-step we impute $q(y)$ for the unlabeled documents only. The M-step computes estimates of μ and $\boldsymbol{\phi}$ from the sum of the observed counts from $\langle \boldsymbol{x}^{(\ell)}, \boldsymbol{y}^{(\ell)} \rangle$ and the expected counts from $\langle \boldsymbol{x}^{(u)} \rangle$ and $q(\boldsymbol{y})$.

Nigam et al. (2000) further parametrize this approach by weighting the unlabeled documents by a scalar λ, which is a tuning parameter. The resulting criterion is:

$$\begin{aligned}\mathcal{L} =& \log \mathrm{p}(\boldsymbol{x}^{(\ell)}, \boldsymbol{y}^{(\ell)}) + \lambda \log \mathrm{p}(\boldsymbol{x}^{(u)}) && (4.24)\\ \geq& \log \mathrm{p}(\boldsymbol{x}^{(\ell)}, \boldsymbol{y}^{(\ell)}) + \lambda E_q[\log \mathrm{p}(\boldsymbol{x}^{(u)}, y)] && (4.25)\end{aligned}$$

The scaling factor does not really have a probabilistic justification, but it can be important to getting good performance, especially when the amount of labeled data is small in comparison to the amount of unlabeled data. In that scenario, the risk is that the unlabeled data will dominate, causing the parameters to drift towards a "natural clustering" that may be a bad fit for the labeled data. Nigam et al. (2000) show that this approach can give substantial improvements in classification performance when the amount of labeled data is small.

Multi-component modeling

Now let us consider an alternative application of EM to supervised classification. One of the classes in 20 newsgroups is `comp.sys.mac.hardware`; suppose that within this newsgroup there are two kinds of posts: reviews of new hardware, and question-answer posts about hardware problems. The language in these **components** of the mac.hardware class might have little in common. So we might do better if we model these components separately. Nigam et al. (2000) show that EM can be applied to this setting as well.

Recall that Naïve Bayes is based on a generative process, which provides a stochastic explanation for the observed data. For multi-component modeling, we envision a slightly different generative process, incorporating both the observed label y_i and the latent component z_i:

- For each document i,
  - draw the label $y_i \sim$ Categorical(μ)
  - draw the component $z_i \mid y_i \sim$ Categorical(β_{y_i}), where $z_i \in 1, 2, \ldots, K_z$.
  - draw the vector of counts $\boldsymbol{x}_i \mid z_i \sim$ Multinomial($\boldsymbol{\phi}_{z_i}$)

Our labeled data includes $\langle \boldsymbol{x}_i, y_i \rangle$, but not z_i, so this is another case of missing data. Again, we sum over the missing data, applying Jensen's inequality to as to obtain a lower bound on the log-likelihood,

$$\begin{aligned}\log \mathrm{p}(\boldsymbol{x}_i, y_i) =& \log \sum_z^{K_z} \mathrm{p}(\boldsymbol{x}_i, y_i, z) && (4.26)\\ \geq& \log \mathrm{p}(y_i; \mu) + E_q \left[\log \mathrm{p}(\boldsymbol{x}_i \mid z; \boldsymbol{\phi}) + \log \mathrm{p}(z \mid y_i; \psi) - \log q_i(z)\right]. && (4.27)\end{aligned}$$

We are now ready to apply expectation-maximization. As usual, the distribution over the missing data — the component z_i — $q_i(z)$ is updated in the E-step. Then during the m-step, we compute:

$$\beta_{y,z} = \frac{E_q\left[\text{count}(y, z)\right]}{\sum_{z'}^{K_z} E_q\left[\text{count}(y, z')\right]} \tag{4.28}$$

$$\boldsymbol{\phi}_{z,j} = \frac{E_q\left[\text{count}(z, j)\right]}{\sum_{j'}^{V} E_q\left[\text{count}(z, j')\right]}. \tag{4.29}$$

Suppose we assume each class y is associated with K components, $\mathcal{Z}_y$. We can then add a constraint to the E-step so that $q_i(z) = 0$ if $z \notin \mathcal{Z}_y \wedge Y_i = y$.

4.4 Other approaches to learning with latent variables*

Expectation maximization is a very general way to think about learning with latent variables, but it has some limitations. One is the sensitivity to initialization, which means that we cannot simply run EM once and expect to get a good solution. Indeed, in practical applications of EM, quite a lot of attention may be devoted to finding a good initialization. A second issue is that EM tends to be easiest to apply in cases where the latent variables have a clear decomposition (in the cases we have considered, they decompose across the instances). For these reasons, it is worth briefly considering some alternatives to EM.

Sampling

Recall that in EM, we set $q(\boldsymbol{z}) = \prod_i q_i(z_i)$, factoring the q distribution into conditionally independent q_i distributions. In sampling-based algorithms, rather than maintaining a distribution over each latent variable, we draw random samples of the latent variables. If the sampling algorithm is designed correctly, this procedure will eventually converge to drawing samples from the true posterior, $\text{p}(\boldsymbol{z}_{1:N} \mid \boldsymbol{x}_{1:N})$. For example, in the case of clustering, we will draw samples from the distribution over clusterings of the data. If a single clustering is required, we can select the one with the highest joint likelihood, $\text{p}(\boldsymbol{z}_{1:N}, \boldsymbol{x}_{1:N})$.

This general family of algorithms is called **Markov Chain Monte Carlo** (MCMC): "Monte Carlo" because it is based on a series of random draws; "Markov Chain" because the sampling procedure must be designed such that each sample depends only on the previous sample, and not on the entire sampling history. Gibbs Sampling is a particularly simple and effective MCMC algorithm, in which we sample each latent variable from its posterior distribution,

$$z_i \mid \boldsymbol{x}, \boldsymbol{z}_{-i} \sim \mathrm{p}(z_i \mid \boldsymbol{x}, \boldsymbol{z}_{-i}), \tag{4.30}$$

where $\boldsymbol{z}_{-i}$ indicates $\{\boldsymbol{z} \backslash z_i\}$, the set of all latent variables except for z_i.

What about the parameters, $\boldsymbol{\phi}$ and μ? One possibility is to turn them into latent variables too, by adding them to the generative story. This requires specifying a prior distribution; the Dirichlet is a typical choice of prior for the parameters of a multinomial, since it has support over vectors of non-negative numbers that sum to one, which is exactly the set of permissible parameters for a multinomial. For example,

$$\boldsymbol{\phi}_y \sim \text{Dirichlet}(\alpha), \forall y \tag{4.31}$$

We can then sample $\boldsymbol{\phi}_y \mid \boldsymbol{x}, \boldsymbol{z} \sim \mathrm{p}(\boldsymbol{\phi}_y \mid \boldsymbol{x}, \boldsymbol{z}, \alpha)$; this posterior distribution will also be Dirichlet, with parameters $\alpha + \sum_{i:y_i=y} \boldsymbol{x}_i$. Alternatively, we can analytically marginalize these parameters, as in **Collapsed Gibbs Sampling**; this is usually preferable if possible. Finally, we might maintain $\boldsymbol{\phi}$ and μ as parameters rather than latent variables. We can employ sampling in the E-step of the EM algorithm, obtaining a hybrid algorithm called Monte Carlo Expectation Maximization (MCEM; Wei and Tanner, 1990).

In principle, these algorithms will eventually converge to the true posterior distribution. However, there is no way to know how long this will take; there is not even any way to check on whether the algorithm has converged. In practice, convergence again depends on initialization, since it might take ages to recover from a poor initialization. Thus, while Gibbs Sampling and other MCMC algorithms provide a powerful and flexible array of techniques for statistical inference in latent variable models, they are not a panacea for the problems experienced by EM.

Murphy (2012) includes an excellent chapter on MCMC; for a more comprehensive treatment, see Robert and Casella (2013).

Spectral learning

A more recent approach to learning with latent variables is based on the **method of moments**. In these approaches, we avoid the problem of non-convex log-likelihood by using a different estimation criterion. Let us write $\overline{\boldsymbol{x}}_i$ for the normalized vector of word counts in document i, so that $\overline{\boldsymbol{x}}_i = \boldsymbol{x}_i / \sum_j x_{ij}$. Then we can form a matrix of word-word co-occurrence counts,

$$\mathbf{C} = \sum_i \boldsymbol{x}_i \boldsymbol{x}_i^\top. \tag{4.32}$$

We can also compute the expected value of this matrix under $\text{p}(\boldsymbol{x} \mid \boldsymbol{\phi}, \mu)$, as

$$\begin{aligned} E[\mathbf{C}] =& \sum_i \sum_k P(Z_i = k \mid \mu) \boldsymbol{\phi}_k \boldsymbol{\phi}_k^\top & (4.33) \\ =& \sum_k N\mu_k \boldsymbol{\phi}_k \boldsymbol{\phi}_k^\top & (4.34) \\ =& \Phi \text{Diag}(N\mu) \Phi^\top, & (4.35) \end{aligned}$$

where Φ is formed by horizontally concatenating $\boldsymbol{\phi}_1 \ldots \boldsymbol{\phi}_K$, and $\text{Diag}(N\mu)$ indicates a diagonal matrix with values $N\mu_k$ at position (k, k). Now, by setting $\mathbf{C}$ equal to its expectation, we obtain,

$$\mathbf{C} = \boldsymbol{\Phi} \text{Diag}(N\mu) \boldsymbol{\Phi}^\top, \tag{4.36}$$

which is very similar to the eigendecomposition $\mathbf{C} = \mathbf{Q}\boldsymbol{\Lambda}\mathbf{Q}^\top$. This suggests that simply by finding the eigenvectors and eigenvalues of $\mathbf{C}$, we could obtain the parameters $\boldsymbol{\phi}$ and μ, and this is what motivates the name **spectral learning**.

However, there is a key difference in the constraints on the solutions to the two problems. In eigendecomposition, we require orthonormality, so that $\mathbf{Q}\mathbf{Q}^\top = \mathbb{I}$. But in estimating the parameters of a mixture model, we require the columns of Φ represents probability vectors, $\forall k, j, \phi_{k,j} \geq 0, \sum_j \phi_{k,j} = 1$, and that the entries of μ correspond to the probabilities over components. Thus, spectral learning algorithms must include a procedure for converting the solution into vectors of probabilities. One approach is to replace eigendecomposition (or the related singular value decomposition) with non-negative matrix factorization (Xu et al., 2003), which guarantees that the solutions are non-negative (Arora et al., 2013).

After obtaining the parameters $\boldsymbol{\phi}$ and μ, we can obtain the distribution over clusters for each document by simply computing $\text{p}(z_i \mid \boldsymbol{x}_i; \boldsymbol{\phi}, \mu) \propto \text{p}(\boldsymbol{x}_i \mid z_i; \boldsymbol{\phi})\text{p}(z_i; \mu)$. The advantages of spectral learning are that it obtains (provably) good solutions without regard to initialization, and that it can be quite fast in practice. Anandkumar et al. (2014) describe how similar matrix and tensor factorizations can be applied to statistical estimation in many other forms of latent variable models.

Chapter 5

Language models

A **language model** is used to compute the probability of a sequence of text. Why would we want to do this? Thus far, we have considered problems where text is the **input**, and we want to select an output, such as a document class or a word sense. But in many of the most prominent problems in language technology, text itself is the output:

- In **machine translation**, we convert from text in a source language to text in a target language.
- In **speech recognition**, we convert audio signal to text.
- In **summarization**, we produce short texts that capture the key points of some longer text.

The goal of language models is to produce more **fluent** text output by computing the probability of the text.

Specifically, suppose we have a vocabulary of word types

$$\mathcal{V} = \{\textit{aardvark}, \textit{abacus}, \ldots, \textit{zither}\} \tag{5.1}$$

Given a sequence of word **tokens**[1] $w_1, w_2, \ldots, w_M$, with every token belonging to some finite vocabulary $\mathcal{V}$, we would like to compute the probability of the

[1] The linguistic term "word" does not cover everything we might want to model, such as names, numbers, and emoticons. Instead, we prefer the term **token**, which refers to anything that can appear in a sequence of linguistic data. **Tokenizers** are programs for segmenting strings of characters or bytes into tokens. In English, tokenization is relatively straightforward, and can be performed using a regular expression. But in languages like Chinese, tokens are not usually separated by spaces, so tokenization can be considerably more challenging. For more on tokenization algorithms, see Manning et al. (2008), chapter 2.

sequence $\mathrm{p}(w_1, w_2, \ldots, w_M)$.[2] We will do this in a data-driven way, assuming we have a **corpus** of text.

So how can we estimate the probability of a sequence of word tokens? The simplest idea would be to apply a **relative frequency estimator**. For example, consider the quote (attributed to Picasso), *Computers are useless, they can only give you answers.* We can estimate the probability of this sentence as follows:

$$\mathrm{p}(\textit{Computers are useless, they can only give you answers}) \tag{5.2}$$

$$= \frac{\text{count}(\textit{Computers are useless, they can only give you answers})}{\text{count}(\text{all sentences ever spoken})} \tag{5.3}$$

It is useful to think about this estimator in terms of bias and variance.

- In the theoretical limit of infinite data, this approach might work. But in practice, we are asking for accurate counts over an infinite number of events, since sequences of words can be arbitrarily long.
- Even if we set an aggressive upper bound of, say, $n = 20$ tokens in the sequence, the number of possible sequences is $\#|\mathcal{V}|^{20}$. A small vocabulary for English would have $\#|\mathcal{V}| = 10^4$, so we would have 10^{80} possible sequences.

Clearly, this estimator is very data-hungry. We need to introduce bias to have a chance of making reliable estimates from finite training data. The language models that follow in this chapter introduce bias in various ways. But before going into detail, let's discuss a little more about the motivation of probabilistic language models.

Are probabilistic language models meaningful? What are the probabilities of the following two sentences?

(5.1) *Colorless green ideas sleep furiously.*

(5.2) *Furiously sleep ideas green colorless.*

Noam Chomsky used this pair of examples to argue that the probability of a sentence is a meaningless concept, from a linguistic standpoint. The reasoning is that any English speaker can tell that the first sentence is grammatical but the second sentence is not. Yet neither sentence, nor their substrings, had ever appeared at the time that Chomsky wrote this article (they have appeared in lots

[2] For now, we'll assume that the vocabulary $\mathcal{V}$ covers all the word tokens that we will ever see. Of course, we can enforce this by allocating a special token ⟨UNK⟩ for unknown words. However, this might not be a great solution, as we will see later.

of linguistics articles since then). Thus, Chomsky argued, empirical probabilities can't distinguish grammatical from ungrammatical sentences.

Pereira (2000) showed that by identifying *classes* of words (e.g., noun, verb, adjective, adverb — but not necessarily these grammatical categories), it is easy to show that the first sentence is more probable than the second. (Class-based language models are discussed in section 5.4.) Do you think this answers Chomsky's argument?

Are probabilistic language models useful? A separate question is whether probabilistic language models are useful for natural language processing. To see how they can help, suppose we want to translate a sentence from Spanish:

(5.3) *El cafe negro me gusta mucho.*

The literal word-for-word translation (sometimes called a **gloss**) is,

(5.4) *The coffee black me pleases much.*

A good language model of English will tell us that the probability of this translation is low. Furthermore,

$$\mathrm{p}(\textit{The coffee black me pleases much}) < \mathrm{p}(\textit{I love dark coffee}). \tag{5.4}$$

How can we use this fact? Warren Weaver, one of the early leaders in machine translation, viewed it as a problem of breaking a secret code (Weaver, 1955):

> When I look at an article in Russian, I say: 'This is really written in English, but it has been coded in some strange symbols. I will now proceed to decode.'

This observation motivates a generative model (like Naïve Bayes!):

- The English sentence $\boldsymbol{w}^{(e)}$ is generated from a language model $\mathrm{p}_{W^e}(\boldsymbol{w}^{(e)})$
- The Spanish sentence $\boldsymbol{w}^{(s)}$ is then generated from a **noisy channel** $\mathrm{p}_{W^s|W^e}(\boldsymbol{w}^{(s)} \mid \boldsymbol{w}^{(e)})$

Our goal is to determine which English sentence might have a generated a given Spanish sentence. This is the **decoding** problem, and is written mathematically as:

$$\max_{\boldsymbol{w}^{(e)}} \mathrm{p}_{W^{(e)}|W^s}(\boldsymbol{w}^{(e)} \mid \boldsymbol{w}^{(s)}) \propto \mathrm{p}_{W^s,W^e}(\boldsymbol{w}^{(s)}, \boldsymbol{w}^{(e)}) = \mathrm{p}_{W^e}(\boldsymbol{w}^{(e)})\mathrm{p}_{W^s|W^e}(\boldsymbol{w}^{(s)} \mid \boldsymbol{w}^{(e)}) \tag{5.5}$$

Probablistic translation in the noisy channel framework incorporates two modular subcomponents:

- The **translation model** is $\mathrm{p}_{W^s|W^e}(\boldsymbol{w}^{(s)} \mid \boldsymbol{w}^{(e)})$. This ensures the **adequacy** of the translation.
- The **language model** is $\mathrm{p}_{W^e}(\boldsymbol{w}^{(e)})$. This ensures the **fluency** of the translation.

It turns out that many phenomena in natural language processing can be viewed in the noisy channel framework. Here are some examples:

- Speech recognition (original = words; encoded = sound)
- Spelling correction (original = well-spelled text; encoded = text with spelling mistakes)
- Summarization (original = summary; encoded = full document)
- Part of speech tagging (original = tags; encoded = words)
- Parsing (original = parse tree; encoded = words)
- Image caption generation (original = caption; encoded = image)

In each case, we solve a decoding problem by converting from the encoded form back to the original that obtains maximum likelihood under the probabilistic model.

A key insight of the noisy channel model is that it allows us to decompose NLP systems into a translation model and a language model, as shown above. Since the language model be estimated from unlabeled data, this means we can improve our system without the expense of obtaining more labeled data — we simply focus on improving $\mathrm{p}_{W^e}(\boldsymbol{w})$. It also means that language models are in principle reusable across many language technology systems. For this reason, I will focus on language models in this chapter, and return to machine translation later in the course.

5.1 N-gram language models

Let us return to the relative frequency estimator,

$$\mathrm{p}(\textit{Computers are useless, they can only give you answers}) \tag{5.6}$$

$$= \frac{\mathrm{count}(\textit{Computers are useless, they can only give you answers})}{\mathrm{count}(\text{all sentences ever spoken})}. \tag{5.7}$$

We will define the probability of a sequence of words as the probability of the words (in order): $\mathrm{p}(\boldsymbol{w}) = \mathrm{p}(w_1, w_2, \ldots, w_M)$. We can apply the chain rule:

$$\begin{aligned}\mathrm{p}(\boldsymbol{w}) =& \mathrm{p}(w_1, w_2, \ldots, w_M)\\ =& \mathrm{p}(w_1)\mathrm{p}(w_2 \mid w_1)\mathrm{p}(w_3 \mid w_2, w_1)\ldots\mathrm{p}(w_M \mid w_{M-1}, \ldots, w_1)\end{aligned}$$

Each element in the product is the probability of a word given all its predecessors. We can think of this as a *word prediction* task: *Computers are [BLANK]*. The relative frequency estimate of the probability of the word *useless* in this context is,

$$\mathrm{p}(\textit{useless} \mid \textit{computers are}) = \frac{\text{count}(\textit{computers are useless})}{\sum_x \text{count}(\textit{computers are } x)} = \frac{\text{count}(\textit{computers are useless})}{\text{count}(\textit{computers are})}.$$

Note that we haven't made any approximations yet, and we could have just as well applied the chain rule in reverse order, $\mathrm{p}(\boldsymbol{w}) = \mathrm{p}(w_M)\mathrm{p}(w_{M-1} \mid w_M)\ldots$, or in any other order. But this means that we also haven't really improved anything either: to compute the conditional probability $P(w_M \mid w_{M-1}, w_{M-2}, \ldots)$, we need to model V^{M-1} contexts, with V possible events. We can't even *store* this probability distribution, let alone reliably estimate it.

N-gram models make a simple approximation: condition on only the past $n-1$ words.

$$\mathrm{p}(w_m \mid w_{m-1} \ldots w_1) \approx P(w_m \mid w_{m-1}, \ldots, w_{m-n+1}) \tag{5.8}$$

This means that the probability of a sentence $\boldsymbol{w}$ can be computed as

$$\mathrm{p}(w_1, \ldots, w_M) \approx \prod_m \mathrm{p}(w_m \mid w_{m-1}, \ldots, w_{m-n+1}) \tag{5.9}$$

To compute the probability of an entire sentence, it is convenient to pad the beginning and end with special symbols $\lozenge$ and $\blacklozenge$. Then the bigram ($n = 2$) approximation to the probability of *I like black coffee* is:

$$\mathrm{p}(\textit{I} \mid \lozenge)\mathrm{p}(\textit{like} \mid \textit{I})\mathrm{p}(\textit{black} \mid \textit{like})\mathrm{p}(\textit{coffee} \mid \textit{black})\mathrm{p}(\blacklozenge \mid \textit{coffee}) \tag{5.10}$$

In this model, we have to estimate and store the probability of only V^n events, which is exponential in the order of the n-gram, and not V^M, which is exponential in the length of the sentence.

The n-gram probabilities can be computed by relative frequency estimation,

$$P(W_i = c \mid W_{i-1} = b, W_{i-2} = a) = \frac{\text{count}(a, b, c)}{\sum_{c'} \text{count}(a, b, c')} = \frac{\text{count}(a, b, c)}{\text{count}(a, b)} \tag{5.11}$$

In estimation, there could be at least two problems with an n-gram language model:

When n is too small. In this case, we are missing important linguistic context. Consider the following sentences:

(5.5) ***Gorillas** always like to groom **THEIR** friends.*

(5.6) *The **computer** that's on the 3rd floor of our office building **CRASHED**.*

The uppercase bolded words depend crucially on their predecessors in lowercase bold: the likelihood of *their* depends on knowing that *gorillas* is plural, and the likelihood of *crashed* depends on knowing that the subject is a *computer*. If the n-grams are not big enough to capture this context, then the resulting language model would offer probabilities that are too low for these sentences, and too high for sentences that fail basic linguistic tests like number agreement.

When n is too big. In this case, we cannot make good estimates of the n-gram parameters from our dataset, because of data sparsity. To handle the *gorilla* example, we would need to model 6-grams; which means accounting for V^6 events. Under a very small vocabulary of $V = 10^4$, this means estimating the probability of 10^{24} distinct events.

These two problems point to another **bias-variance** tradeoff. Can you see how it works? In practice, we often have **both** problems at the same time. Language is full of long-range dependencies that we cannot capture because n is too small; at the same time, language datasets are full of rare phenomena, whose probabilities we fail to estimate accurately because n is too large.

We will seek approaches to keep n large, while still making low-variance estimates of the underlying parameters. To do this, we will introduce a different sort of bias: **smoothing**. But before we talk about that, let's consider how we can evaluate language models.

5.2 Evaluating language models

Because language models are typically components of larger systems — language modeling is not really an application itself — we would prefer **extrinsic evaluation**. This means evaluating whether the language model improves performance on the application task, such as machine translation or speech recognition. But this is often hard to do, and depends on details of the overall system which may be irrelevant to language modeling. In contrast, **intrinsic evaluation** is task-neutral.

Better performance on intrinsic metrics may be expected to improve extrinsic metrics across a variety of tasks, unless we are over-optimizing the intrinsic metric. We will discuss intrinsic metrics here, but bear in mind that it is important to also perform extrinsic evaluations to ensure that the improvements obtained on these intrinsic metrics really carry over to the applications that we care about.

Held-out likelihood

A popular intrinsic metric is the **held-out likelihood**. To compute this metric, we "hold out" a portion of our data from training. We compute the log probability of this held-out data, according to the model that we estimate from the training set. A good language model should assign high probability to this held-out data. Specifically, we compute,

$$\ell(\boldsymbol{w}) = \sum_{i}^{N} \sum_{m}^{M_i} \log \mathrm{p}(w_m^{(i)} \mid w_{m-1}^{(i)}, \ldots, w_{m-n+1}^{(i)}), \tag{5.12}$$

summing over all sentences $\{\boldsymbol{w}^{(i)}\}_{i \in 1 \ldots N}$ in the held-out corpus.

Typically, unknown words in the test data are mapped to the $\langle$UNK$\rangle$ token. This means that we have to estimate some probability for $\langle$UNK$\rangle$ on the training data. One way to do this is to fix the vocabulary $\mathcal{V}$ to the $V - 1$ words with the highest counts in the training data, and then convert all other tokens to $\langle$UNK$\rangle$.

Perplexity

Perplexity is a transformation of the held-out likelihood into an information-theoretic quantity. Specifically, we compute

$$PP(\boldsymbol{w}) = 2^{-\frac{\ell(\boldsymbol{w})}{M}}, \tag{5.13}$$

where M is the total number of tokens in the held-out corpus.

- The transformation means that lower perplexities correspond to higher likelihoods, so lower scores are better on this metric. (Lower perplexity is better, because you are less perplexed.) In the limit, we obtain probability 1 for our held-out corpus, with $PP = 2^{-\log 1} = 2^0 = 1$.

- Assume a uniform, unigram model in which $\text{p}(w_i) = \frac{1}{V}$ for all V words in the vocabulary. Then,

$$\begin{aligned} PP(\boldsymbol{w}) &= \left[\left(\frac{1}{V}\right)^M\right]^{-\frac{1}{M}} \\ &= \left(\frac{1}{V}\right)^{-1} = V \end{aligned}$$

These observations imply that we can think of perplexity as the *weighted branching factor* at each word in the sentence.

- If we have solved the word prediction problem perfectly, $PP(\boldsymbol{w}) = 1$, because there is only one possible choice for each word.
- If we have a uniform model that assigns equal probability to every word, then $PP(\boldsymbol{w}) = V$. This is not a worst-case scenario — in the worst case, we assign **zero** probability to some word in the test data — but it is a worst "reasonable" case.
- Most models give perplexities that fall somewhere in between 1 and V.

Example On 38M tokens of WSJ, $V \approx 20K$, (Jurafsky and Martin, 2009, page 97) obtain these perplexities on a 1.5M token test set.

- Unigram: 962
- Bigram: 170
- Trigram: 109

Will it keep going down?

Information theory*

Perplexity is closely related to the concept of **entropy**, the expected value of the information contained in each word.

$$H(P) = -\sum_{\boldsymbol{w}} \text{p}(\boldsymbol{w}) \log \text{p}(\boldsymbol{w}) \tag{5.14}$$

The true entropy of English (or any real language) is unknown. Claude Shannon, one of the founders of information theory, wanted to compute upper and lower bounds. He would read passages of 15 characters to his wife, and ask her to guess the next character, recording the number of guesses it took for her to get the correct answer. As a fluent speaker of English, his wife could provide a reasonably tight bound on the number of guesses needed per character.[3]

We can view the goal of language modeling as computing a distribution Q that is similar to the true distribution P. To measure the quality of Q, we can compute its **cross-entropy** with P, written as $H(P, Q)$,

$$\begin{aligned}
H(P, Q) =& E_P[\log Q] && (5.15)\\
=& -\sum_{\boldsymbol{w}} \mathrm{p}(\boldsymbol{w}) \log q(\boldsymbol{w}) && (5.16)\\
=& -\sum_{\boldsymbol{w}} \mathrm{p}(\boldsymbol{w}) \log \left(q(\boldsymbol{w}) \frac{\mathrm{p}(\boldsymbol{w})}{\mathrm{p}(\boldsymbol{w})} \right) && (5.17)\\
=& -\sum_{\boldsymbol{w}} \mathrm{p}(\boldsymbol{w}) \log \frac{q(\boldsymbol{w})}{\mathrm{p}(\boldsymbol{w})} + \mathrm{p}(\boldsymbol{w}) \log \mathrm{p}(\boldsymbol{w}) && (5.18)\\
=& \sum_{\boldsymbol{w}} \mathrm{p}(\boldsymbol{w}) \log \frac{\mathrm{p}(\boldsymbol{w})}{q(\boldsymbol{w})} - \mathrm{p}(\boldsymbol{w}) \log \mathrm{p}(\boldsymbol{w}) && (5.19)\\
=& D_{KL}(P||Q) + H(P), && (5.20)
\end{aligned}$$

where $D_{KL}(P||Q)$ is the **Kullback-Leibler (KL) divergence** between P and Q. The KL-divergence is a non-symmetric measure of the the dissimilarity of two distributions, where $\forall (P, Q), D_{KL}(P||Q) \geq 0$ and $D_{KL}(P||P) = 0$.[4] The cross-entropy also includes a term for the entropy of the true distribution P, but since P is given, we can only control Q. Thus, minimizing the cross entropy $H(P, Q)$ is equivalent to minimizing the KL-divergence $D_{KL}(P||Q)$.

We do not have access to the true P, just a sequence $\boldsymbol{w} = \{w_1, w_2, \ldots, \}$, which is sampled from P. In the limit, the length of $\boldsymbol{w}$ is infinite, so we have,

$$\begin{aligned}
H(P, Q) =& -\sum_{\boldsymbol{w}} \mathrm{p}(\boldsymbol{w}) \log q(\boldsymbol{w}) && (5.21)\\
=& -\lim_{M \to \infty} \frac{1}{M} \log q(\boldsymbol{w}). && (5.22)
\end{aligned}$$

[3] Question for you: is this an upper bound or a lower bound?

[4] KL-divergence has connections to expectation maximization: the lower bound on the expected likelihood can be viewed as the true likelihood minus the KL-divergence $D_{KL}(q(\boldsymbol{y})||\mathrm{p}(\boldsymbol{y} \mid \boldsymbol{x}))$, so that the E-step minimizes the KL-divergence by setting $q(\boldsymbol{y}) = \mathrm{p}(\boldsymbol{y} \mid \boldsymbol{x})$.

There term $\mathrm{p}(\boldsymbol{w})$ disappears because the word sequence $\boldsymbol{w}$ is itself a sample from this distribution. In practice, we have finite M, so we compute the approximation,

$$H(P,Q) \approx -\frac{1}{M}\log q(\boldsymbol{w}) \tag{5.23}$$

$$PP(Q) = 2^{-\frac{1}{M}\log q(\boldsymbol{w})} = 2^{H(P,Q)} \tag{5.24}$$

Thus, the perplexity of the language model Q can be derived from its cross-entropy with the true word distribution P, which we estimate from the observed word sequence $\boldsymbol{w}$. Low perplexity implies low cross-entropy, which in turn implies a low KL-divergence between P and Q.

Further aside A related topic in psycholinguistics is the "constant entropy rate hypothesis," also called the "uniform information density hypothesis." The hypothesis is that speakers should prefer linguistic choices that convey a uniform amount of information over time (Jaeger, 2010). Some evidence:

- Speakers shorten predictable words, and lengthen unpredictable ones (Jaeger, 2010).
- Low-probability words slow down the reader (Smith and Levy, 2013)
- Syntactic reductions (e.g., *I'm* versus *I am*) are more likely when the reducible word contains less information (Jaeger and Levy, 2006).

5.3 Smoothing and discounting

Limited data is a persistent problem in estimating language models. In section 5.1, we presented n-grams as a partial solution. But as we saw, sparse data can be a problem even for low-order n-grams; at the same time, many linguistic phenomena, like subject-verb agreement, cannot be incorporated into language models without higher-order n-grams. It is therefore necessary to add additional inductive biases to n-gram language models. This section covers some of the most intuitive and common approaches, but there are many more. Chen and Goodman (1999) provides a good survey of the state-of-the-art in the late 1990s; more recent approaches are discussed in section 5.4.

Smoothing

A major concern in language modeling is to avoid the situation $\mathrm{p}(\boldsymbol{w}) = 0$, which could arise as a result of a single unseen n-gram. A similar problem arose in

Naïve Bayes, and there we solved it by **smoothing**: adding pseudo counts. The same idea can be applied to n-gram language models, as shown here in the bigram case,

$$\mathrm{p}_{\text{Laplace}}(W_i = b \mid W_i = a) = \frac{\text{count}(a, b) + \alpha}{\sum_{w'} \text{count}(a, w') + V\alpha}. \tag{5.25}$$

- In general, this is called **Lidstone smoothing**.
- When $\alpha = 1$, it is **Laplace smoothing**.
- When $\alpha = 0.5$, we are following **Jeffreys-Perks law**.
- Manning and Schütze (1999) offer more insight on the justifications for Jeffreys-Perks smoothing

To maintain normalization, anything that we add to the numerator (α) must also appear in the dominator ($V\alpha$). This idea is reflected in the concept of **effective counts**:

$$c_i^* = (c_i + \alpha)\frac{N}{N + V\alpha}, \tag{5.26}$$

where c_i is the count of event i, and c_i^* is the effective count. The **discount** for each n-gram is then computed as,

$$d_i = \frac{c_i^*}{c_i} = \frac{(c_i + \alpha)}{c_i}\frac{N}{(N + \alpha)}$$

Discounting and backoff

Discounting "borrows" probability mass from observed n-grams and redistributes it. In Lidstone smoothing, we borrow probability mass by increasing the denominator of the relative frequency estimates, and redistribute it by increasing the numerator for all n-grams. But instead, we could borrow the same amount of probability mass from all observed counts, and redistribute it among only the unobserved counts. This is called **absolute discounting.**

For example, if we set an absolute discount $d = 0.1$ in a trigram model, we get: $\mathrm{p}(w \mid \textit{denied the}) =$

We need not redistribute the probability mass equally. Instead, we can **backoff** to a lower-order language model. In other words: if you have trigrams, use trigrams; if you don't have trigrams, use bigrams; if you don't even have bigrams,

word	counts c	effective counts c^*	unsmoothed probability	smoothed probability
allegations	3	2.9	0.429	0.414
reports	2	1.9	0.286	0.271
claims	1	0.9	0.143	0.129
request	1	0.9	0.143	0.129
charges	0	0.2	0.000	0.029
benefits	0	0.2	0.000	0.029
...				

use unigrams. This is called **Katz backoff**:

$$c^*(u,v) = c(u,v) - d \tag{5.27}$$

$$\text{p}_{\text{backoff}}(v \mid u) = \begin{cases} \frac{c^*(u,v)}{c(u)} & \text{if } c(u,v) > 0 \\ \alpha(u) \times \frac{\text{p}_{\text{backoff}}(v)}{\sum_{v' : c(u,v')=0} \text{p}_{\text{backoff}}(v')} & \text{if } c(u,v) = 0 \end{cases} \tag{5.28}$$

Typically we can set the discount d to minimize perplexity on a development set.

Interpolation

An alternative to discounting is **interpolation**: setting the probability of a word in context to a weighted sum of its probabilities across progressively shorter contexts.

Instead of choosing a single n for the size of the n-gram, we can take the weighted average across several n-gram probabilities,

$$\begin{aligned} \text{p}_{\text{Interpolation}}(a \mid b, c) = {} & \lambda_3^{(a)} \text{p}_3^*(a \mid b, c) \\ & + \lambda_2^{(a)} \text{p}_2^*(a \mid b) \\ & + \lambda_3^{(a)} \text{p}_1^*(a). \end{aligned}$$

In this equation, p_k^* is the maximum likelihood estimate (MLE) of a k-gram model, and $\lambda_k^{(a)}$ is the weight of the n-gram model p_k^* for word a. A nice property of this model is that it can learn to use longer context for some words (e.g., possessive pronouns like *his* and *her*, which often match the gender of the entity as defined earlier in the sentence), and shorter context for others (e.g., rare content words).

To ensure that the interpolated $\text{p}(\boldsymbol{w})$ is still a probability, we have a constraint, $\sum_k \lambda_k^{(a)} = 1, \forall a$. But how to find the specific values of λ for each word? An elegant solution is **expectation maximization**. Recall from chapter 4 that we can think about EM as learning with **missing data**: we just need to choose missing data such that learning would be easy if it weren't missing. So what's missing in this case? We can think of each word w_m as drawn from an n-gram of unknown size, $z_m \in \{1 \ldots n\}$. This z_m is the missing data that we are looking for! Specifically, consider the following generative story:

- For each token m,
  - draw $z_m \sim \text{Categorical}(\lambda^{(w_m)})$
  - draw $w_m \sim \text{p}^*_{z_m}(w_m \mid w_{m-1}, \ldots w_{m-z_m})$.

If we knew $\{z_m\}_{m \in 1 \ldots M}$, then we could compute λ from relative frequency estimation, $\lambda^{(a)}_k = \frac{\sum_m \delta(z_m=k)\delta(w_m=a)}{\sum_m \delta(w_m=a)}$.[5] Since we do not know the values of the missing data, we impute a distribution $q_m(z_m)$ in the E-step, which represents our degree of belief that word token w_m was generated from a n-gram of order z_m.

Having defined these quantities, we can derive EM updates:

- **E-step**:

$$\begin{aligned} q_m(k) =& P(Z_m = k \mid \boldsymbol{w}_{1:m}) && (5.29) \\ & \propto \text{p}^*_z(w_m \mid w_{m-1}, \ldots, w_{m-k+1}) \lambda^{(w_m)}_k && (5.30) \end{aligned}$$

- **M-step**:

$$\begin{aligned} \lambda_k(a) =& \frac{E_q\left[\text{count}(W = a, Z = k)\right]}{\sum_{k'} E_q\left[\text{count}(W = a, Z = k')\right]} && (5.31) \\ =& \frac{\sum_m q_m(k)\delta(w_m = a)}{\sum_m \delta(w_m = a)} && (5.32) \end{aligned}$$

As usual, EM iterates between these two steps until convergence to a local optimum.

Kneser-Ney smoothing

Kneser-Ney smoothing also incorporates discounting, but redistributes the resulting probability mass in a different way. Consider the example: *I recently visited*

[5] We could also use z to update our n-gram models p^*_i, but we will assume those are fixed here.

- *Francisco*?
- *Duluth*?

Now suppose that both bigrams *visited Duluth* and *visited Francisco* are unobserved in our training data, and furthermore, that the unigram probability $\mathrm{p}^*(\textit{Francisco})$ is greater than $\mathrm{p}^*(\textit{Duluth})$. Nonetheless we would still guess that $\mathrm{p}(\textit{visited Duluth}) > P(\textit{visited Francisco})$, because *Duluth* is a more **versatile** word. This notion of versatility is the key to Kneser-Ney smoothing.

Writing c for a context of undefined length, and $\text{count}(w, c)$ as the count of word w in context c, we define the Kneser-Ney bigram probability as

$$\mathrm{p}_{KN}(w \mid c) = \begin{cases} \frac{\text{count}(w,c)-d}{\text{count}(c)}, & \text{count}(w,c) > 0 \\ \alpha(c)\mathrm{p}_{\text{continuation}}(w), & \text{otherwise} \end{cases}$$

$$\mathrm{p}_{\text{continuation}}(w) = \frac{\#|c : \text{count}(w,c) > 0|}{\sum_{w'} \#|c' : \text{count}(c', w') > 0|}$$

First, note that we reserve probability mass using absolute discounting d, which is taken from all unobserved n-grams. The total amount of discounting in context c is $d \times \#|w : \text{count}(w,c) > 0|$, and we divide this equally among the unseen n-grams,

$$\alpha(c) = \frac{d \times \#|w : \text{count}(w,c) > 0|}{\#|c : \text{count}(w,c) = 0|}. \tag{5.33}$$

This is the amount of probability mass left to account for versatitility, which we define via the *continuation probability* $\mathrm{p}_{\text{continuation}}(w)$ as proportional to the number of observed contexts in which w appears. In the numerator of the continuation probability we have the number of contexts c in which w appears, and in the denominator, we normalize by computing the same quantity over all words w'.

In practice, interpolation works a little better than backoff,

$$\mathrm{p}_{KN}(w \mid c) = \frac{\text{count}(w,c) - d}{\text{count}(c)} + \lambda_c \mathrm{p}_{\text{continuation}}(w) \tag{5.34}$$

This idea of counting contexts may seem heuristic, but there is a cool theoretical justification from Bayesian nonparametrics (Teh, 2006).

5.4 Other language models

Interpolated Kneser-Ney is pretty close to state-of-the-art. But there are some interesting other types of language models, and they apply ideas that we have already learned.

Class-based language models

The reason we need smoothing is because even the trigram probability model $P(w_m, \mid w_{m-1}, w_{m-2})$ has a huge number of parameters. We could use the idea of **word classes** to simplify. Imagine that each word has a latent class z,

$$\mathrm{p}_{\text{class}}(w_m \mid w_{m-1}) = \sum_z \mathrm{p}(w_m \mid z; \boldsymbol{\phi})\mathrm{p}(z \mid w_{m-1}; \beta),$$

where $z \in [1, K], K \ll V$. This means that each word w_m is conditioned on its class z through parameter $\boldsymbol{\phi}_z$, and the class itself is conditioned on the previous word w_{m-1} through parameter $\beta_{w_{m-1}}$. The advantage of this approach is that it gives a bigram probability using $2 \times V \times K$ parameters, instead of V^2.

How do we estimate such a model? Since there is missing data — the word classes — we might use expectation maximization:

- E-step: update $q_i(z)$
- M-step: update $\boldsymbol{\phi}$ and β

But this is usually too slow in practice, since it requires multiple passes over the training data, which is typically very large. A useful approximate algorithm is **exchange clustering** (Brown et al., 1992), which assigns each word type to a single class, rather than maintaining a soft distribution $\mathrm{p}(z \mid w; \beta)$. This algorithm incrementally constructs a binary tree over the words in the vocabulary, so that each word can be represented by a bit vector corresponding to the series of left/right decisions to get to the word from the root. The prefixes of these bit vectors are an early form of **word embedding**, and it has been shown that syntactically similar words tend to have similar bit vectors, as shown in Figure 5.1. As we will see in chapter 20, these vectors can be used as features in NLP systems, improving their performance by enabling generalization from frequent to rare words (Miller et al., 2004; Koo et al., 2008).

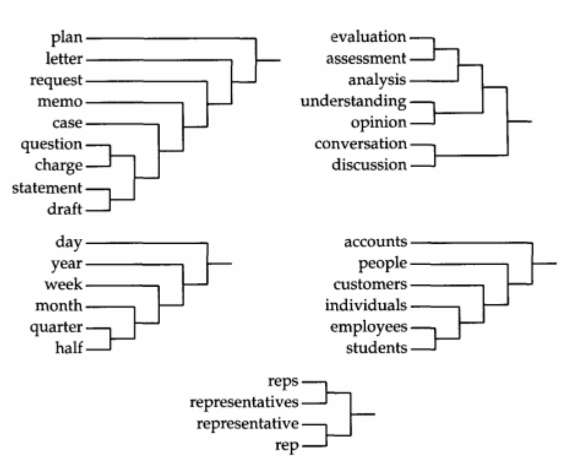


Figure 5.1: Example subtrees from the Brown et al. (1992) hierarchical class-based language model

Discriminative language models

Alternatively, we could just train a model to predict $\text{p}(w_m \mid w_{m-1}, w_{m-2}, \ldots)$ directly. We can think of this as a straightforward classification problem, where the label space is equal to the entire vocabulary; for example, Rosenfeld (1996) applies logistic regression to language modeling, and Roark et al. (2007) apply perceptrons and conditional random fields (section 6.5). A key advantage is that discriminative training minimizes the error rate, rather than maximizing probability; for applications such as speech recognition, this is a better fit for the ultimate goal, which is recognizing speech with as few errors as possible. Moreover, because the underlying model is now discriminative, additional features can be included, such as features of the syntactic structure (Khudanpur and Wu, 2000). However, Roark et al. (2007) report that discriminative language models are expensive to train, requiring extensive feature selection.

Neural language models

Currently, **neural probabilistic language models** are attracting a lot of interest. These are related to discriminative language models, but they also maintain a continuous state that can capture long-term history. Another key distinction in these models is the use of dense, discriminatively-trained vector representations, computing the probability $\mathrm{p}(w_m \mid \boldsymbol{w}_{1:m-1})$ by passing an inner product $\boldsymbol{h}_m^\top \boldsymbol{v}_{w_{m-1}}$ through a sigmoid activation function.[6]

There are many flavors of neural language models, from the early formulation by Bengio et al. (2003) to more elaborate contemporary models based on long short-term memory (LSTM; Hochreiter and Schmidhuber, 1997; Sundermeyer et al., 2012). We will focus on the Recurrent Neural Network Language Model (RNNLM; Mikolov et al., 2010), which works fairly well in practice, and is the basis for some of the more complex recent models.

Assume each word type i is associated with a dense vector representation $\boldsymbol{u}_i$, which is a parameter of the model. Writing $\boldsymbol{x}_m$ is an indicator vector such that

$$x_{mi} = \begin{cases} 1, & i = w_m \\ 0, & i \neq w_m, \end{cases} \tag{5.35}$$

then we can write $\boldsymbol{u}_{w_m} = \mathbf{U}\boldsymbol{x}_m$. In fact, we will have two dense vector representations per word: $\mathbf{U}$ for the input and $\mathbf{V}$ for the output. We will return to word vectors in chapter 15 and chapter 21.

The RNNLM is **recurrent** in the sense that there is a hidden state $\boldsymbol{h}_m$ at each word position m, which is constructed from the hidden state $\boldsymbol{h}_{m-1}$, as well as from the word w_m. Specifically, the hidden state in the RNNLM is given by,

$$\boldsymbol{h}_m = f(\mathbf{U}\boldsymbol{x}_m + \Theta \boldsymbol{h}_{m-1}), \tag{5.36}$$

where f is an element-wise non-linear activation function, such as the sigmoid. Finally, we predict w_{m+1} with probability

$$\mathrm{p}(w_{m+1} \mid \boldsymbol{h}_m) = \frac{\exp\left(\boldsymbol{h}_m^\top \boldsymbol{v}_{w_{m+1}}\right)}{\sum_i \exp\left(\boldsymbol{h}_m^\top \boldsymbol{v}_i\right)}. \tag{5.37}$$

Since the hidden states $\boldsymbol{h}_{1:m}$ can be computed deterministically from the words $\boldsymbol{w}_{1:m}$, the RNNLM defines a distribution $\mathrm{p}(w_{m+1} \mid \boldsymbol{w}_{1:m})$ without any explicit limit on the length of the past history. However, information from words $w_j : j \ll m$ will be attenuated by repeatedly passing through the recurrent function. Recent variants on the RNNLM address this issue through the use of memory cells (Sundermeyer et al., 2012) and gates (Chung et al., 2015), enabling crucial pieces of past information to more directly impact future predictions.

The RNNLM has three parameters: the word representation matrices $\mathbf{U}$ and $\mathbf{V}$, and the recurrent update matrix Θ. Note that the size of these parameters are

[6] The function $p_{ij} = \frac{\exp \psi_{ij}}{\sum_{j'} \exp \psi_{ij'}}$ is sometimes called **softmax** in the neural net literature.

relatively small. Writing K for the size of each word vector representation (assuming the input and output representations have identical size), the parameter dimensions are $K \times V$ for $\mathbf{U}$ and $\mathbf{V}$, and $K \times K$ for $\mathbf{\Theta}$. Considering that $V \gg K$ in all practical cases, this means that the RNNLM has far fewer parameters than even a bigram language model, which must score counts of size $V \times V$. For this reason, neural language models require less attention to smoothing and regularization than traditional n-gram language models.

The RNNLM parameters are learned by **backpropagation** from a loss function: a typical choice is the negative log-likelihood of the data, which is identical to the **cross-entropy**:

$$\ell = \sum_{m} \log \mathrm{p}(w_m \mid \boldsymbol{w}_{1:m-1}) \tag{5.38}$$

$$= \sum_{m} \boldsymbol{h}_{m-1}^{\top} \boldsymbol{v}_{w_m} - \log \sum_{i}^{V} \exp\left(\boldsymbol{h}_{m-1}^{\top} \boldsymbol{v}_i\right). \tag{5.39}$$

Computing this loss function (and its gradients) can be expensive, since it involves summing over the entire vocabulary at each word position. One alternative is to use a **hierarchical softmax** function to compute the sum more efficiently, in $\log V$ time (Mikolov et al., 2011); another is to optimize an alternative metric, such as noise-contrastive estimation, which learns by distinguishing observed instances from artificial instances generated from a noise distribution (Mnih and Teh, 2012).

Recent work on **probabilistic programming** has resulted in a number of toolkits for building computation graphs over architectures such as the RNNLM. These toolkits — Theano and Torch are currently popular choices — perform automatic differentiation, allowing the researcher to plug in a variety of different loss functions and model architectures, without having to derive and implement the parameter updates by hand.

5.5 Other details

Datasets Dataset genre is important: an LM learned from Shakespeare is a poor match for the Wall Street Journal (WSJ); an LM learned from the WSJ is a poor choice for predictive text entry in cellphones.

Vocabulary We have assumed we know the total vocabulary size V. Will we always know this? What if we don't?

- Suppose we are estimating a bigram language model. Then we can set $V_{bigram} = V^2_{unigram}$, assuming we have seen all unigrams.
- But in general, we're always at risk of seeing new words (`http://www.americandialect.org/hashtag-2012`)
    - *hashtag* (2012 word of the year)
    - *phablet*
    - *Gangnam*[7]
    - *-3.78109932019384*
- If the set of unigrams is defined in advance, this is the **closed vocabulary** setting. Typically we will just replace unknown words with a special token, ⟨unk⟩.
- Another solution is to backoff from the unigram model to a character model:

$$P_u(s_i) = \begin{cases} \frac{\text{count}(s_i)}{\text{count(all tokens)}+\beta}, & s_i \in \mathcal{V} \\ \beta P_c(s_i), & s_i \notin \mathcal{V} \end{cases} \tag{5.40}$$

$$P_c(s_i) = P_{\text{len}}(s_i) \prod_{a_j \in s_i} \frac{\text{count}(a_j)}{\text{count(all characters)}} \tag{5.41}$$

  [todo: reconcile this notation with the rest of the chapter] We could even have a bigram or trigram model over characters.
- Still another possibility, this time in the setting of neural language models, would be to require that word representations are themselves composed from a character-level RNNLM (Ling et al., 2015). Besides ensuring that we can always compute a word representation for any sequence of character symbols, this approach implies that similarly-spelled words have similar representations.

[7] [todo: Wow this list is already incredibly dated! I should probably try to use more timeless examples, like *Sputnik* or something.]

Part II

Sequences and trees

Chapter 6

Sequence labeling

In sequence labeling, we want to assign tags to words, or more generally, we want to assign discrete labels to elements in a sequence. There are many applications of sequence labeling in natural language processing, and chapter 7 presents an overview. One of the most classic application of sequence labeling is **part-of-speech tagging**, which involves tagging each word by its grammatical category. Coarse-grained grammatical categories include NOUNs, which describe things, properties, or ideas, and VERBs, which describe actions and events. Given a simple sentence like,

(6.1) They can fish.

we would like to produce the tag sequence N V V, with the modal verb *can* labeled as a verb in this simplified example.

6.1 Sequence labeling as classification

One way to solve tagging problems is to treat them as classification. A simple tagging model would have a single base feature, the word itself:

$$\boldsymbol{f}(\boldsymbol{w} = \textit{they can fish}, \mathrm{N}, 1) = \langle \textit{they}, \mathrm{N} \rangle \tag{6.1}$$

$$\boldsymbol{f}(\boldsymbol{w} = \textit{they can fish}, \mathrm{V}, 2) = \langle \textit{can}, \mathrm{V} \rangle \tag{6.2}$$

$$\boldsymbol{f}(\boldsymbol{w} = \textit{they can fish}, \mathrm{V}, 3) = \langle \textit{fish}, \mathrm{V} \rangle . \tag{6.3}$$

Here the feature function takes three arguments as input: the sentence to be tagged (*they can fish* in all cases), the proposed tag (e.g., N or V), and the word token to which this tag is applied. This simple feature function then returns a single feature: a tuple including the word to be tagged and the tag that has been proposed.

If the vocabulary size is V and the number of tags is K, then there are $V \times K$ features. Each of these features must be assigned a weight. These weights can be learned from a labeled dataset using a classification algorithm such as perceptron, but this isn't necessary in this case: it would be equivalent to define the classification weights directly from a dictionary, with $\theta_{w,y} = 1$ for the best tag y for each word w, and $\theta_{w,y} = 0$ for all other tags.

However, it is easy to see that this simple classification approach can go wrong. Consider the word *fish*, which often describes the animal rather than the activity; in these cases, *fish* should be tagged as a noun. To tag ambiguous words correctly, the tagger must rely on context, such as the surrounding words. We can build this context into the feature set by incorporating the surrounding words as additional features:

$$\begin{aligned}
\boldsymbol{f}(\boldsymbol{w} = \textit{they can fish}, \mathrm{N}, 1) = \{&\langle w_i = \textit{they}, y_i = \mathrm{N}\rangle, \\
&\langle w_{i-1} = \Diamond, y_i = \mathrm{N}\rangle, \\
&\langle w_{i+1} = \textit{can}, y_i = \mathrm{N}\rangle\} && (6.4) \\
\boldsymbol{f}(\boldsymbol{w} = \textit{they can fish}, \mathrm{V}, 2) = \{&\langle w_i = \textit{can}, y_i = \mathrm{V}\rangle, \\
&\langle w_{i-1} = \textit{they}, y_i = \mathrm{V}\rangle, \\
&\langle w_{i+1} = \textit{fish}, y_i = \mathrm{V}\rangle\} && (6.5) \\
\boldsymbol{f}(\boldsymbol{w} = \textit{they can fish}, \mathrm{V}, 3) = \{&\langle w_i = \textit{fish}, y_i = \mathrm{V}\rangle, \\
&\langle w_{i-1} = \textit{can}, y_i = \mathrm{V}\rangle, \\
&\langle w_{i+1} = \blacklozenge, y_i = \mathrm{V}\rangle\}. && (6.6)
\end{aligned}$$

These features contain enough information that a tagger should be able to choose the right label for the word *fish*: words that follow the modal verb *can* are likely to be verbs themselves, so the feature $\langle w_{i-1} = \textit{can}, y_i = \mathrm{V}\rangle$ should have a large positive weight.

However, even with this enhanced feature set, it may be difficult to tag some sequences correctly. One reason is that there are often relationships between the tags themselves. For example, in English it is relatively rare for a verb to follow another verb — particularly if we differentiate **M**ODAL verbs like *can* and *should* from more typical verbs, like *give*, *transcend*, and *befuddle*. We would like to incorporate preferences **against** such tag sequences, and preferences **for** other tag sequences, such as NOUN-VERB.

The need for such preferences is best illustrated by a **garden path sentence**:

(6.2) The old man the boat.

Grammatically, the word *the* is a DETERMINER. When you read the sentence, what part of speech did you first assign to *old*? Typically, this word is an ADJEC-

TIVE — abbreviated as J — which is a class of words that modify nouns. Similarly, *man* is usually a noun. The resulting sequence of tags is D J N D N. But this is a mistaken "garden path" interpretation, which ends up leading nowhere. It is unlikely that a determiner would directly follow a noun,[1] and particularly unlikely that the entire sentence would lack a verb. The only possible verb in the sentence is the word *man*, which can refer to the act of maintaining and piloting something — often boats. But if *man* is tagged as a verb, then *old* is seated between a determiner and a verb, and must be a noun. And indeed, adjectives can often have a second interpretation as nouns when used in this way (e.g., *the young*, *the restless*). This reasoning, in which the labeling decisions are intertwined, cannot be applied in a setting where each tag is produced by an independent classification decision.

6.2 Sequence labeling as structure prediction

As an alternative, we can think of the entire sequence of tags as a label itself. For a given sequence of words $\boldsymbol{w}_{1:M} = (w_1, w_2, \ldots, w_M)$, there is a set of possible taggings $\mathcal{Y}(\boldsymbol{w}_{1:M}) = \mathcal{Y}^M$, where $\mathcal{Y} = \{\text{N}, \text{V}, \text{D}, \ldots\}$ refers to the set of individual tags, and $\mathcal{Y}^M$ refers to the set of tag sequences of length M. We can then treat the sequence labeling problem as a classification problem in this exponential label space,

$$\hat{\boldsymbol{y}}_{1:M} = \underset{\boldsymbol{y}_{1:M} \in \mathcal{Y}(\boldsymbol{w}_{1:M})}{\text{argmax}} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}_{1:M}, \boldsymbol{y}_{1:M}), \tag{6.7}$$

where $\boldsymbol{y}_{1:M} = (y_1, y_2, \ldots, y_M)$ is a sequence of M tags. Note that in this formulation, we have a feature function that consider the entire tag sequence $\boldsymbol{y}_{1:M}$. Such a feature function can therefore include features that capture the relationships between tagging decisions, such as the preference that determiners not follow nouns, or that all sentences have verbs.

Is it possible to perform tagging in this way? The problem of making a series of interconnected labeling decisions is known as **inference**. Because natural language is full of interrelated grammatical structures, inference is a crucial aspect of contemporary natural language processing. Can we perform inference over label sequences? In English, it is not unusual to have sentences of length $M = 20$; part-of-speech tag sets vary in size from 10 to several hundred. Taking the low end of this range, we have $\#|\mathcal{Y}(\boldsymbol{w}_{1:M})| \approx 10^{20}$, one hundred billion billion possible tag sequences. Enumerating and scoring each of these sequences would require

[1] The main exception is the double object construction, as in *I gave the child a toy.*

an amount of work that is exponential in the sequence length; in other words, inference is intractable.

However, the situation changes when we restrict the feature function. Suppose we choose features that never consider more than one tag. We can indicate this restriction as,

$$\boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}) = \sum_{m}^{M} \boldsymbol{f}(\boldsymbol{w}, y_m, m), \tag{6.8}$$

meaning that the overall feature vector is the sum of feature vectors associated with individual tagging decisions. Such features are not capable of capturing the intuitions that might help us solve garden path sentences, such as the insight that determiners rarely follow nouns in English. But this restriction does make it possible to find the globally optimal tagging, by making a sequence of individual tagging decisions.

$$\begin{aligned}
\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}) =& \boldsymbol{\theta}^\top \sum_{m} \boldsymbol{f}(\boldsymbol{w}, y_m, m) && (6.9)\\
=& \sum_{m} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y_m, m) && (6.10)\\
\hat{y}_m =& \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y_m, m) && (6.11)\\
\hat{\boldsymbol{y}} =& (\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_M) && (6.12)
\end{aligned}$$

Note that we are still searching over an exponentially large set of tag sequences! But the feature set restriction results in decoupling the labeling decisions that were previous interconnected. As a result, it is not necessary to score every one of the M^K tag sequences individually — we can find the optimal sequence by scoring the local parts of these decisions.

Now let's consider a slightly less restrictive feature function: rather than considering only individual tags, we will consider adjacent tags too. This means that we can have negative weights for infelicitous tag pairs, such as noun-determiner, and positive weights for typical tag pairs, such as determiner-noun and noun-verb. We define this feature function as,

$$\boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}) = \sum_{m}^{M} \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m). \tag{6.13}$$

Let's apply this feature function to the shorter example, *they can fish*, using features for word-tag and tag-tag pairs:

$$
\begin{aligned}
\boldsymbol{f}(\boldsymbol{w} = \textit{they can fish}, \boldsymbol{y} = \text{N V V}) = & \sum_{m=1}^{M} \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m) && (6.14)\\
= & \boldsymbol{f}(\boldsymbol{w}, \text{N}, \Diamond, 1)\\
& + \boldsymbol{f}(\boldsymbol{w}, \text{V}, \text{N}, 2)\\
& + \boldsymbol{f}(\boldsymbol{w}, \text{V}, \text{V}, 3) && (6.15)\\
= & \langle w_m = \textit{they}, y_m = \text{N}\rangle + \langle y_m = \text{N}, y_{m-1} = \Diamond\rangle\\
& + \langle w_m = \textit{can}, y_m = \text{V}\rangle + \langle y_m = \text{V}, y_{m-1} = \text{N}\rangle\\
& + \langle w_m = \textit{fish}, y_m = \text{V}\rangle + \langle y_m = \text{V}, y_{m-1} = \text{V}\rangle\\
& + \langle y_m = \blacklozenge, y_{m-1} = \text{V}\rangle && (6.16)
\end{aligned}
$$

We end up with seven active features: one for each word-tag pair, and one for each tag-tag pair (this includes a final tag $y_{M+1} = \blacklozenge$). These features capture what are arguably the two main sources of information for part-of-speech tagging: which tags are appropriate for each word, and which tags tend to follow each other in sequence. Given appropriate weights for these features, we can expect to make the right tagging decisions, even for difficult cases like *the old man the boat*.[2]

The example shows that even with the restriction to the feature set shown in Equation 6.13, it is still possible to construct expressive features that are capable of solving many sequence labeling problems. But the key question is: does this restriction make it possible to perform efficient inference? The answer is yes, and the solution is the Viterbi algorithm Viterbi (1967).

6.3 The Viterbi algorithm

We now consider the inference problem,

$$\hat{\boldsymbol{y}} = \operatorname*{argmax}_{\boldsymbol{y}} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}) \tag{6.17}$$

$$\boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}) = \sum_{m=1}^{M} \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m). \tag{6.18}$$

Given this restriction on the feature function, we can solve this inference problem using **dynamic programming**, a algorithmic technique for reusing work in

[2] [todo: Question: given these features, what inequality in the weights must hold for the correct tag sequence to outscore the garden path tag sequence for this example.]

recurrent computations. As in many dynamic programming problems, we begin by solving an auxiliary problem: rather than finding the best tag sequence, we simply try to compute the **score** of the best tag sequence,

$$\begin{aligned}
\max_{\boldsymbol{y}} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}) =& \max_{\boldsymbol{y}_{1:M}} \sum_{m=1}^{M} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m) && (6.19)\\
=& \max_{\boldsymbol{y}_{1:M}} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y_M, y_{M-1}, M) + \sum_{m=1}^{M-1} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m) && (6.20)\\
=& \max_{y_M} \max_{y_{M-1}} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y_M, y_{M-1}, M) + \max_{\boldsymbol{y}_{1:M-2}} \sum_{m=1}^{M-1} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m). && (6.21)
\end{aligned}$$

In this derivation, we first removed the final element $\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y_M, y_{M-1}, M)$ from the sum over the sequence, and then we adjusted the scope of the the $\max$ operation, since the elements $(y_1 \ldots y_{M-2})$ are irrelevant to the final term.

Let us now define the **Viterbi variable**,

$$v_m(y) \triangleq \max_{\boldsymbol{y}_{1:m-1}} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y, y_{m-1}, m) + \sum_{n=1}^{m-1} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y_n, y_{n-1}, n), \tag{6.22}$$

where lower-case m indicates any position in the sequence, and y indicates a tag for that position. This variable represents the score of the best tag sequence $(\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m)$ that terminates in $\hat{y}_m = y$. From this definition, we can compute the score of the best tagging of the sequence by plugging the Viterbi variables $v_M(\cdot)$ into Equation 6.21,

$$\max_{\boldsymbol{y}} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}) = \max_{k} v_M(k). \tag{6.23}$$

Now, let us look more closely at how we can compute these Viterbi variables.

$$
\begin{aligned}
v_m(y) \triangleq & \max_{\boldsymbol{y}_{1:m-1}} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y, y_{m-1}, m) + \sum_{n=1}^{m-1} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y_n, y_{n-1}, n) && (6.24)\\
= & \max_{y_{m-1}} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y, y_{m-1}, m) \\
& + \max_{\boldsymbol{y}_{1:m-2}} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y_{m-1}, y_{m-2}) + \sum_{n=1}^{m-2} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y_n, y_{n-1}, n) && (6.25)\\
= & \max_{y_{m-1}} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y, y_{m-1}, m) + v_{m-1}(y_{m-1}) && (6.26)\\
v_1(y) = & \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y, \Diamond, 1). && (6.27)
\end{aligned}
$$

Equation 6.26 is a **recurrence** for computing the Viterbi variables: each $v_m(y)$ can be computed in terms of $v_{m-1}(\cdot)$, and so on. We can therefore step forward through the sequence, computing first all variables $v_1(\cdot)$ from Equation 6.27, and then computing all variables $v_2(\cdot)$, $v_3(\cdot)$, and so on, until we reach the final set of variables $v_M(\cdot)$.

Graphically, it is customary to arrange these variables graphically with the sequence index m on the rows, and the tag index y on the columns; in this representation, each $v_{m-1}(y)$ is connected to each $v_m(y)$, forming a **trellis**. This is shown in Figure 6.1. As shown in the figure, we set aside special nodes for the start and end states.

Our real goal is to find the best scoring sequence, not simply to compute its score. But as is often the case in dynamic programming, solving the auxiliary program gets us almost all the way to our original goal. Recall that each $v_m(k)$ represents the score of the best tag sequence ending in $Y_m = k$. To compute this, we maximize over possible values of Y_{m-1}. If we keep track of the tag that maximizes this choice at each step, then we can walk backwards from the final tag, and recover the optimal tag sequence. This is indicated in Figure 6.1 by the solid blue lines, which we trace back from the final position.

Why does this work? We can make an inductive argument. Suppose k is indeed the optimal tag for word m, and we now need to decide on the tag Y_{m-1}. Because we make the inductive assumption that we know $Y_m = k$, and because the feature function is restricted to adjacent tags, we need not consider any of the tags $\boldsymbol{Y}_{m+1:M}$; these tags, and the features that describe them, are irrelevant to the inference of Y_{m-1}, given the choice of $Y_m = k$. Thus, we are looking for the tag

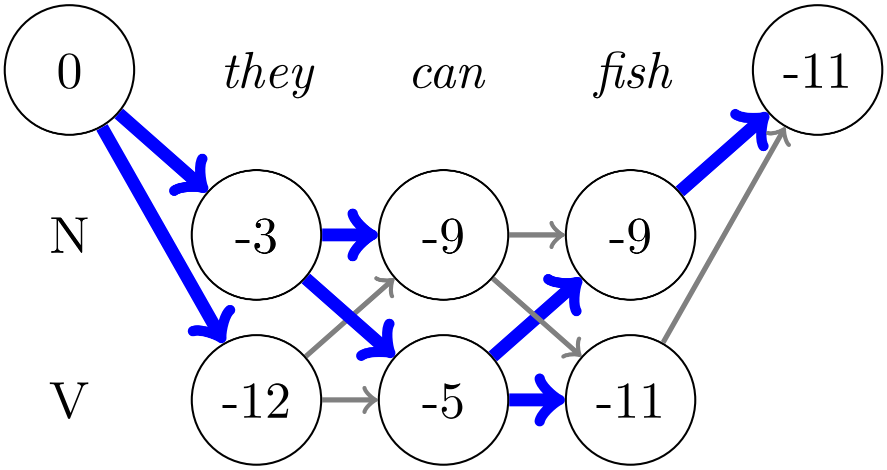


Figure 6.1: The trellis representation of the Viterbi variables, for the example *They can fish.*

Algorithm 5 The Viterbi algorithm.

```
for $k \in \{0, \dots K\}$ do
    $v_1(k) = \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, k, \Diamond, m)$
for $m \in \{2, \dots, M\}$ do
    for $k \in \{0, \dots, K\}$ do
        $v_m(k) = \max_{k'} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, k, k', m) + v_{m-1}(k')$
        $b_m(k) = \text{argmax}_{k'} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, k, k', m) + v_{m-1}(k')$
$y_M = \text{argmax}_k v_M(k) + \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, \blacklozenge, k, M+1)$
for $m \in \{M-1, \dots 1\}$ do
    $y_m = b_m(y_{m+1})$
```

$\hat{y}_{m-1}$ that maximizes,

$$\hat{y}_{m-1} = \underset{y_{m-1}}{\text{argmax}}\, \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, Y_m = k, Y_{m-1} = y_{m-1}, m) + \max_{\boldsymbol{y}_{1:m-2}} \sum_{n=1}^{m-1} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y_n, y_{n-1}, n) \tag{6.28}$$

$$= \underset{y_{m-1}}{\text{argmax}}\, \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, Y_m = k, y_{m-1}, m) + v_{m-1}(y_{m-1}), \tag{6.29}$$

which we obtain by plugging in the definition of the Viterbi variable.

	they	*can*	*fish*
N	-2	-3	-3
V	-10	-1	-3

(a) Weights for "emission" features.

	N	V	♦
◊	-1	-2	$-\infty$
N	-3	-1	-2
V	-1	-3	-2

(b) Weights for "transition" features.

Table 6.1: Feature weights for the example trellis shown in Figure 6.1. Emission weights from $\Diamond$ and $\blacklozenge$ are implicitly set to $-\infty$. [todo: double check that this example works.]

The complete Viterbi algorithm is shown in Algorithm 5. This formalizes the recurrences that were described in the previous two paragraphs, and handles the boundary conditions at the start and end of the sequence. Specifically, when computing the initial Viterbi variables $v_1(\cdot)$, we use a special tag, $\Diamond$, to indicate the start of the sequence. When computing the final tag Y_M, we use another special tag, $\blacklozenge$, to indicate the end of the sequence. These special tags enable the use of transition features for the tags that begin and end the sequence: for example, nouns are relatively likely to start part-of-speech sequences in English, so we would like to have a positive weight for the feature $\langle \Diamond, \mathrm{N} \rangle$; conjunctions are unlikely to end sequences, so we would like a negative weight for the feature $\langle \mathrm{CC}, \blacklozenge \rangle$.

What is the complexity of this algorithm? If there are K tags and M positions in the sequence, then there are $M \times K$ Viterbi variables to compute. To compute each variable, we must compute a maximum over K possible predecessor tags. The total computation cost of populating the trellis is therefore $\mathcal{O}(MK^2)$, with an additional factor for the number of active features at each position. After completing the trellis, we simply trace the backwards pointers to the beginning of the sequence, which takes $\mathcal{O}(M)$ operations.

Example

We now illustrate the Viterbi algorithm with an example, using the minimal tagset $\{\mathrm{N}, \mathrm{V}\}$, corresponding to nouns and verbs. Even in this tagset, there is considerable ambiguity: consider the words *can* and *fish*, which can each take both tags. The sentence *They can fish* therefore has four possible taggings, two of which are grammatical. The tagging *They*/N *can*/V *fish*/N corresponds to the scenario of putting fish into cans.)

To begin, we use the feature weights defined in Table 6.1. These weights are used to incrementally fill in the trellis. As described in Algorithm 5, we fill in the cells from left to right, with each column corresponding to a word in the sequence. As we fill in the cells, we must keep track of the "back-pointers" $b_m(k)$ — the previous cell that maximizes the score of tag k at word m. These are represented in the figure with the thick blue lines. At the end of the algorithm, we recover the optimal tag sequence by tracing back the optimal path from the final position, $(M+1, k = \blacklozenge)$.

Higher-order features

The Viterbi algorithm was made possible by a restriction of the features to consider only pairs of adjacent tags. In a sense, we can think of this as a bigram language model, at the tag level. A natural question is how we might generalize Viterbi to tag trigrams:

$$\boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}) = \sum_{m}^{M} \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, y_{m-2}, m). \tag{6.30}$$

One possibility is to take the Cartesian product of the tagset with itself, $\mathcal{Y}^{(2)} = \mathcal{Y} \times \mathcal{Y}$. The tags in this product space are ordered pairs, representing adjacent tags at the token level: for example, the tag $\langle \text{N}, \text{V} \rangle$ would represent a noun followed by a verb. Transitions between such tags must be consistent: we can have a transition from $\langle \text{N}, \text{V} \rangle$ to $\langle \text{V}, \text{N} \rangle$ (corresponding to the token-level tag sequence N V N), but not from $\langle \text{N}, \text{V} \rangle$ to $\langle \text{N}, \text{N} \rangle$, which would not correspond to any token-level tag sequence. This constraint can be enforced in the feature weights, with $\theta_{\langle \langle a,b \rangle, \langle c,d \rangle \rangle} = -\infty$ if $b \neq c$. The remaining feature weights can encode preferences for and against various tag trigrams.

The extension to trigrams changes the time and space complexity of the Viterbi algorithm. Recall that the time complexity of the Viterbi algorithm is $\mathcal{O}(MK^2)$: the size of the trellis is $M \times K$, and computing each element of the trellis requires maximizing over K possible predecessors. Now, the Cartesian product tag set is quadratic in the size of the original tag set, so the size of the trellis must be $M \times K^2$, with K representing the size of the original tagset. A naïve implementation of Viterbi in the Cartesian product tag space would compute each element of the trellis by maximizing over K^2 possible predecessors, for a total time complexity of $\mathcal{O}(MK^4)$. But this is more work than is necessary: even though the size of the tag-pair space is K^2, there are still only K possible predecessors for each position in the trellis, due to the consistency constraints mentioned in the previous paragraph. Careful design of the $\max$ operation in the Viterbi algorithm can exploit this, limiting the time complexity to $\mathcal{O}(MK^3)$. In general, the time and space complexity of the Viterbi algorithm grows exponentially with the order of the tag n-grams.

6.4 Hidden Markov Models

We now consider how to learn the weights $\boldsymbol{\theta}$ that parametrize the Viterbi sequence labeling algorithm. We begin with a probabilistic approach. Recall that the probabilistic Naïve Bayes classifier selects the label y to maximize $\mathrm{p}(y \mid \boldsymbol{x}) \propto \mathrm{p}(y, \boldsymbol{x})$. In probabilistic sequence labeling, our goal is similar: select the tag sequence that maximizes $\mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w}) \propto \mathrm{p}(\boldsymbol{y}, \boldsymbol{w})$. Just as Naïve Bayes could be cast as a linear classifier maximizing $\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}, y)$, we can cast our probabilistic classifier as a linear decision rule. Furthermore, the feature restriction in Equation 6.13 can be viewed as an Markov independence assumption on the elements of $\boldsymbol{y}$. Thanks to this assumption, it is possible to perform inference using the Viterbi algorithm.

The Naïve Bayes algorithm was introduced as a generative model — a probabilistic story that explains the observed data as well as the hidden label. A similar story can be constructed for probabilistic sequence labeling: first, we draw the tags from a prior distribution, $\boldsymbol{y} \sim \mathrm{p}(\boldsymbol{y})$; next, we draw the tokens from a conditional likelihood distribution, $\boldsymbol{w} \sim \mathrm{p}(\boldsymbol{w} \mid \boldsymbol{y})$. However, for inference to be tractable, additional independence assumptions are required. Here we make two assumptions. First, the probability of each token depends only on its tag, and not on any other element in the sequence:

$$\mathrm{p}(\boldsymbol{w} \mid \boldsymbol{y}) = \prod_{m}^{M} \mathrm{p}(w_m \mid y_m). \tag{6.31}$$

Next, we introduce an independence assumption on the form of the prior distribution over labels: each label y_m depends only on its predecessor,

$$\mathrm{p}(\boldsymbol{y}) = \prod_{m=1}^{M} \mathrm{p}(y_m \mid y_{m-1}), \tag{6.32}$$

where $y_0 = \lozenge$ in all cases. Due to this **Markov** assumption, probabilistic sequence labeling models are known as **hidden Markov models** (HMMs). We now state the generative model under these independence assumptions,

$$\begin{tikzcd}
y_1 \arrow[r] \arrow[d] & y_2 \arrow[r] \arrow[d] & \cdots \arrow[r] & y_M \arrow[d] \\
w_1 & w_2 & \ldots & w_M
\end{tikzcd}$$

Figure 6.2: Graphical representation of the hidden Markov model. Arrows indicate probabilistic dependencies.

- For $m \in (1, 2, \ldots, M)$,
  - draw $y_m \mid y_{m-1} \sim$ Categorical$(\boldsymbol{\lambda}_{y_{m-1}})$;
  - draw $w_m \mid y_m \sim$ Categorical$(\boldsymbol{\phi}_{y_m})$

This generative story formalizes the hidden Markov model. Given the parameters $\boldsymbol{\lambda}$ and $\boldsymbol{\phi}$, we can compute $\mathrm{p}(\boldsymbol{w}, \boldsymbol{y})$ for any token sequence $\boldsymbol{w}$ and tag sequence $\boldsymbol{y}$. The HMM is often represented as a **graphical model** (Wainwright and Jordan, 2008), as shown in Figure 6.2. This representation makes the independence assumptions explicit: if a variable x is probabilistically conditioned on another variable y, then there is an arrow $x \rightarrow y$ in the diagram; otherwise, x and y are **conditionally independent**, given the other variables in the model.

It is important to reflect on the implications of the HMM independence assumptions. A non-adjacent pair of tags y_m and y_n are conditionally independent; if $m < n$ and we are given y_{n-1}, then y_m offers no additional information about y_m. However, if we are not given any information about the tags in a sequence, then all tags are probabilistically coupled.

Estimation

The hidden Markov model has two groups of parameters:

Emission probabilities. The probability $\mathrm{p}_e(w_m \mid y_m; \phi)$ is the emission probability, since the words are treated as probabilistically "emitted", conditioned on the tags.

Transition probabilities. The probability $\mathrm{p}_t(y_m \mid y_{m-1}; \lambda)$ is the transition probability, since it assigns probability to each possible tag-to-tag transition.

Both of these groups of parameters are typically computed from relative frequency estimation on a labeled corpus,

$$\begin{aligned}
\phi_{k,i} \triangleq& \mathrm{p}(W_m = i \mid Y_m = k) = \frac{\text{count}(W_m = i, Y_m = k)}{\text{count}(Y_m = k)}\\
\lambda_{k,k'} \triangleq& \mathrm{p}(Y_m = k' \mid Y_{m-1} = k) = \frac{\text{count}(Y_m = k', Y_{m-1} = k)}{\text{count}(Y_{m-1} = k)}.
\end{aligned}$$

Smoothing is more important for the emission probability than the transition probability, because the event space is much larger. Smoothing techniques such as additive smoothing, interpolation, and backoff (see chapter 5) can all be applied here.

Inference

The goal of inference in the hidden Markov model is to find the highest probability tag sequence,

$$\hat{\boldsymbol{y}} = \operatorname*{argmax}_{\boldsymbol{y}} \mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w}). \tag{6.33}$$

As in Naïve Bayes, it is equivalent to find the tag sequence with the highest **log-probability**, since the log function is monotonically increasing. It is furthermore equivalent to maximize the joint probability $\mathrm{p}(\boldsymbol{y}, \boldsymbol{w}) = \mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w})\mathrm{p}(\boldsymbol{w}) \propto \mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w})$, which is proportional to the conditional probability. Therefore, we can reformulate the inference problem as,

$$\hat{\boldsymbol{y}} = \operatorname*{argmax}_{\boldsymbol{y}} \log \mathrm{p}(\boldsymbol{y}, \boldsymbol{w}). \tag{6.34}$$

We can now apply the HMM independence assumptions:

$$\begin{aligned}
\log \mathrm{p}(\boldsymbol{y}, \boldsymbol{w}) =& \log \mathrm{p}(\boldsymbol{y}) + \log \mathrm{p}(\boldsymbol{w} \mid \boldsymbol{y}) && (6.35)\\
=& \sum_{m=1}^{M} \log \mathrm{p}_y(y_m \mid y_{m-1}) + \log \mathrm{p}_{w|y}(w_m \mid y_m) && (6.36)\\
=& \sum_{m=1}^{M} \log \lambda_{y_m, y_{m-1}} + \log \phi_{y_m, w_m}. && (6.37)
\end{aligned}$$

This log probability can be rewritten as a dot product of weights and features,

$$\begin{aligned}\log \mathrm{p}(\boldsymbol{y}, \boldsymbol{w}) = & \sum_{m=1}^{M} \log \lambda_{y_m, y_{m-1}} + \log \phi_{y_m, w_m} && (6.38)\\ = & \sum_{m=1}^{M} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m), && (6.39)\end{aligned}$$

where the feature function $\boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m) = \{\langle y_m, y_{m-1}\rangle, \langle y_m, w_m\rangle\}$, and the weight vector θ encodes the log-parameters $\log \boldsymbol{\lambda}$ and $\log \boldsymbol{\phi}$.

This derivation shows that HMM inference can be viewed as an application of the Viterbi decoding algorithm, given an appropriately defined feature function and weight vector. The local product $\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m)$ can be interpreted probabilistically,

$$\begin{aligned}\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m) = & \log \mathrm{p}_y(y_m \mid y_{m-1}) + \log \mathrm{p}_{w|y}(w_m \mid y_m) && (6.40)\\ = & \log \mathrm{p}(y_m, w_m \mid y_{m-1}) && (6.41)\end{aligned}$$

Now recall the definition of the Viterbi variables,

$$\begin{aligned}v_m(y) = & \max_{\boldsymbol{y}_{1:m-1}} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y, y_{m-1}, m) + \sum_{n=1}^{m-1} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y_n, y_{n-1}, n) && (6.42)\\ v_m(y) = & \max_{\boldsymbol{y}_{1:m-1}} \log \mathrm{p}(y_m, w_m \mid y_{m-1}) + \sum_{n=1}^{m-1} \log \mathrm{p}(y_n, w_n \mid y_{n-1}) && (6.43)\\ = & \max_{\boldsymbol{y}_{1:m-1}} \log \mathrm{p}(\boldsymbol{y}_{1:m-1}, Y_m = y, \boldsymbol{w}_{1:m}). && (6.44)\end{aligned}$$

In words, the Viterbi variable $v_m(y)$ is the log probability of the best tag sequence ending in $Y_m = y$, joint with the word sequence $\boldsymbol{w}_{1:m}$. The log probability of the best complete tag sequence is therefore,

$$\max_{\boldsymbol{y}_{1:M}} \log \mathrm{p}(\boldsymbol{y}_{1:M}, \boldsymbol{w}_{1:M}) = \max_{y_M} \log \mathrm{p}_y(\blacklozenge \mid y_M) + v_M(y_M). \qquad (6.45)$$

The Viterbi algorithm can be implemented using probabilities, rather than log probabilities. In this case, each $v_m(y)$ is equal to,

$$\begin{aligned} v_m(y) &= \max_{\boldsymbol{y}_{1:m-1}} \mathrm{p}(\boldsymbol{y}_{1:m-1}, Y_m = y, \boldsymbol{w}_{1:m}) && (6.46)\\ &= \max_{y_{m-1}} \mathrm{p}(y_m, w_m \mid y_{m-1}) \max_{\boldsymbol{y}_{1:m-2}} \mathrm{p}(y_{1:m-2}, y_{m-1}, \boldsymbol{w}_{1:m-1}) && (6.47)\\ &= \max_{y_{m-1}} \mathrm{p}(y_m, w_m \mid y_{m-1}) \times v_{m-1}(y_{m-1}) && (6.48)\\ &= \mathrm{p}(w_m \mid y_m) \times \max_{y_{m-1}} \mathrm{p}(y_m \mid y_{m-1}) \times v_{m-1}(y_{m-1}) && (6.49)\\ &= \max_{y_{m-1}} \exp(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m)) \times v_{m-1}(y_{m-1}). && (6.50) \end{aligned}$$

In the final line, we use the fact that $\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m) = \log \mathrm{p}(y_m, w_m \mid y_{m-1})$, and exponentiate the dot product to obtain the probability.

In practice, the probabilities tend towards zero over long sequences, so the log-probability version of Viterbi is more practical from the standpoint of numerical stability. However, this version of the algorithm is often taught first, since it can be explained directly in terms of probabilities. It also connects to a broader literature on inference in graphical models. Each Viterbi variable is computed by **maximizing** over a set of **products**; thus, the Viterbi algorithm is a special case of the **max-product algorithm** for inference in graphical models (Wainwright and Jordan, 2008).

The Forward Algorithm

In an influential survey, Rabiner (1989) defines three problems for hidden Markov models:

Decoding Find the best tags $\boldsymbol{y}$ for a sequence $\boldsymbol{w}$.

Likelihood Compute the marginal probability $\mathrm{p}(\boldsymbol{w}) = \sum_{\boldsymbol{y}} \mathrm{p}(\boldsymbol{w}, \boldsymbol{y})$.

Learning Given only unlabeled data $\{\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots, \boldsymbol{w}_D\}$, estimate the transition and emission distributions.

The Viterbi algorithm solves the decoding problem. We'll talk about the learning problem in section 6.6. Let's now consider how to compute the marginal likelihood $\mathrm{p}(\boldsymbol{w}) = \sum_{\boldsymbol{y}} \mathrm{p}(\boldsymbol{w}, \boldsymbol{y})$, which involves summing over all possible tag sequences. There are at least two reasons we might want to do this:

Language modeling Note that the probability $\mathrm{p}(\boldsymbol{w})$ is also computed by the language models that were discussed in chapter 5. In those language models, we used only unlabeled corpora, conditioning each token w_m on previous tokens. An HMM-based language model would leverage a corpus of part-of-speech annotations, and therefore might be expected to generalize better than an n-gram language model — for example, it would be more likely to assign positive probability to a nonsense grammatical sentence like *colorless green ideas sleep furiously.*

Tag marginals It is often important to compute marginal probabilities of individual tags, $\text{p}(y_m \mid \boldsymbol{w}_{1:M})$. This is the probability distribution over tags for token m, conditioned on all of the words $\boldsymbol{w}_{1:M}$. For example, we might like the know the probability that a given word is tagged as a verb, regardless of how all the other words are tagged. We will discuss how to compute this probability in **??**, but as a preview, we will use the following form,

$$\text{p}(y_m \mid \boldsymbol{w}_{1:M}) = \frac{\text{p}(y_m, \boldsymbol{w}_{1:M})}{\text{p}(\boldsymbol{w}_{1:M})}, \tag{6.51}$$

which involves the marginal likelihood in the denominator.

We can compute the marginal likelihood using a dynamic program that is nearly identical to the Viterbi algorithm. We will use probabilities for now, and show the conversion to log-probabilities later. The core of the algorithm is to compute a set of **forward variables**,

$$\alpha_m(y) \triangleq \text{p}(Y_m = y, \boldsymbol{w}_{1:m}). \tag{6.52}$$

From this definition, we can compute the marginal likelihood by summing over the final forward variables,

$$\text{p}(\boldsymbol{w}) = \sum_y \text{p}(Y_M = y, \boldsymbol{w}_{1:M}) \tag{6.53}$$

$$= \sum_y \alpha_M(y). \tag{6.54}$$

To capture the probability of terminating the sequence on each possible tag Y_M, we can pad the end of $\boldsymbol{w}$ with an extra token ■, which can only be emitted from the stop tag ♦.

The forward variables themselves can be computed recursively,

$$
\begin{aligned}
\alpha_m(k) =& \Pr(Y_m = k, \boldsymbol{W}_{1:m} = \boldsymbol{w}_{1:m}) && (6.55)\\
=& \mathrm{p}(w_m \mid Y_m = k) \times \Pr(Y_m = k \mid \boldsymbol{w}_{1:m-1}) && (6.56)\\
=& \mathrm{p}(w_m \mid Y_m = k) \times \sum_{k'} \Pr(Y_m = k, Y_{m-1} = k' \mid \boldsymbol{w}_{1:m-1}) && (6.57)\\
=& \mathrm{p}(w_m \mid Y_m = k) \times \sum_{k'} \Pr(Y_m = k \mid Y_{m-1} = k') \times \Pr(Y_{m-1} = k' \mid \boldsymbol{w}_{1:m-1}) && (6.58)\\
=& \mathrm{p}(w_m \mid Y_m = k) \times \sum_{k'} \Pr(Y_m = k \mid Y_{m-1} = k') \times \alpha_{m-1}(k') && (6.59)\\
=& \sum_{k'} \exp(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}_{1:M}, Y_m = k, Y_{m-1} = k', m)) \times \alpha_{m-1}(k'). && (6.60)
\end{aligned}
$$

The derivation relies on the independence assumptions in the hidden Markov model: W_m depends only on Y_m, and Y_m is conditionally independent from $W_{1:m-1}$ and all tags, given Y_{m-1}. We complete the derivation by introducing Y_{m-1} and summing over all possible values, and by then applying the chain rule to obtain the final recursive form.

Procedurally, we compute the forward variables in just the same way as we compute the Viterbi variables: we first compute all $\alpha_1(\cdot)$, then all $\alpha_2(\cdot)$, and so on. We initialize each $\alpha_0(k) = \mathrm{p}(Y_m = k \mid Y_{m-1} = \Diamond)$, to capture the transition probability from the start symbol. Comparing Equation 6.59 to Equation 6.49, the sole difference is that instead of maximizing over possible values of Y_{m-1}, we sum. Just as the Viterbi algorithm is a special case of the max-product algorithm for inference in graphical models, the forward algorithm is a special case of the **sum-product** algorithm for computing marginal likelihoods.

In practice, it is numerically more stable to compute the marginal log-probability. In the log domain, the forward recurrence is,

$$
\begin{aligned}
\alpha_m(k) \triangleq& \log \mathrm{p}(\boldsymbol{w}_{1:m}, Y_m = k) && (6.61)\\
=& \log \sum_{k'} \exp(\log \mathrm{p}(w_m \mid Y_m = k) + \log \Pr(Y_m = k \mid Y_{m-1} = k')\\
&+ \log \mathrm{p}(\boldsymbol{w}_{1:m-1}, Y_{m-1} = k')) && (6.62)\\
=& \log \sum_{k'} \exp(\log \mathrm{p}(w_m \mid Y_m = k) + \log \Pr(Y_m = k \mid Y_{m-1} = k')\\
&+ \alpha_{m-1}(k')) && (6.63)\\
=& \log \sum_{k'} \exp(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}_{1:M}, Y_m = k, Y_{m-1} = k', m) + \alpha_{m-1}(k')). && (6.64)
\end{aligned}
$$

Scientific programming libraries provide numerically robust implementations of the log-sum-exp function, which should prevent overflow and underflow from exponentiation.

Semiring Notation and the Generalized Viterbi Algorithm

We have now seen the Viterbi and Forward recurrences, each of which can be performed over probabilities or log probabilities. These four recurrences are closely related, and can in fact be expressed as a single recurrence in a more general notation, known as **semiring algebra**. We use the symbol $\oplus$ to represent generalized addition, and the symbol $\otimes$ to represent generalized multiplication.[3] Given these operators, we can denote a generalized Viterbi recurrence as,

$$v_m(k) = \bigoplus_{k'} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, Y_m = k, Y_{m-1} = k', m) \otimes v_{m-1}(k'). \tag{6.65}$$

Each recurrence that we have seen so far is a special case of this generalized Viterbi recurrence:

- In the max-product Viterbi recurrence over probabilities, the $\oplus$ operation corresponds to maximization, and the $\otimes$ operation corresponds to multiplication.
- In the Forward recurrence over probabilities, the $\oplus$ operation corresponds to addition, and the $\otimes$ operation corresponds to multiplication.
- In the max-product Viterbi recurrence over log-probabilities, the $\oplus$ operation corresponds to maximization, and the $\otimes$ operation corresponds to addition. (This is sometimes called the **tropical semiring**, in honor of the Brazilian mathematician Imre Simon.)
- In the Forward recurrence over log-probabilities, the $\oplus$ operation corresponds to log-addition, $a \oplus b = \log(e^a + e^b)$. The $\otimes$ operation corresponds to addition.

The mathematical abstraction offered by semiring notation can be applied to the software implementations of these algorithms, yielding concise and modular implementations. The OPENFST library (Allauzen et al., 2007) is an example of a software package in which the algorithms are parametrized by the choice of semiring.

[3] In a semiring, the addition and multiplication operators must both obey associativity, and multiplication must distribute across addition; the addition operator must be commutative; there must be additive and multiplicative identities $\overline{0}$ and $\overline{1}$; and there must be a multiplicative annihilator $\overline{0}$, such that $a \otimes \overline{0} = \overline{0}$.

6.5 Discriminative sequence labeling

Today, hidden Markov models are rarely used for supervised sequence labeling. This is because HMMs are limited to only two phenomena:

- Word-tag probabilities, via the emission probability $\mathrm{p}_E(w_m \mid y_n)$;
- local context, via the transition probability $\mathrm{p}_T(y_m \mid y_{m-1})$.

However, as we have seen, the Viterbi algorithm can be applied to much more general feature sets, as long as the decomposition $\boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}) = \sum_{m=1}^{M} \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m)$ is observed. In this section, we discuss methods for learning the weights on such features. However, let's first pause to ask what additional features might be needed.

Word affix features. Consider the problem of part-of-speech tagging on the first four lines of the poem *Jabberwocky* (Carroll, 1917):

(6.3) 'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe:
All mimsy were the borogoves,
And the mome raths outgrabe.

Many of these words are made up, so you would have no information about their probabilities of being associated with any particular part of speech. Yet it is not so hard to see what their grammatical roles might be in this passage. Context helps: for example, the word *slithy* follows the determiner *the*, and therefore is likely to be a noun or adjective. Which do you think is more likely? The suffix *-thy* is found in a number of adjectives — e.g., *frothy,healthy,pithy,worthy*. The suffix is also found in a handful of nouns — e.g., *apathy,sympathy* — but nearly all of these nouns contain *-pathy*, unlike *slithy*. The suffix gives some evidence that *slithy* is an adjective, and indeed it is: later in the text we find that it is a combination of the adjectives *lithe* and *slimy*.[4]

Fine-grained context. Another useful source of information is fine-grained context — that is, contextual information that is more specific than the previous tag.

[4] **Morphology** is the study of how words are formed from smaller linguistic units. Computational approaches to morphological analysis are touched on in chapter 8; Bender (2013) provides a good overview of the underlying linguistic principles.

For example, consider the noun phrases *this fish* and *these fish*. Many part-of-speech tagsets distinguish between singular and plural nouns, but do not distinguish between singular and plural determiners; for example, the Penn Treebank tagset follows these conventions. A hidden Markov model would be unable to correctly label *fish* as singular or plural in both of these cases, because it only has access to two features: the preceding tag (determiner in both cases) and the word (*fish* in both cases). The classification-based tagger discussed in section 6.1 had the ability to use preceding and succeeding words as features, and we would like to incorporate this information into a sequence labeling algorithm.

Example Suppose we have the tagging D J N (determiner, adjective, noun) for the sequence *the slithy toves* in Jabberwocky, so that

$$\begin{aligned}\boldsymbol{w} =& \textit{the slithy toves}\\ \boldsymbol{y} =& \text{D J N}.\end{aligned}$$

We now create the feature vector for this example, assuming that we have word-tag features (indicated by prefix W), tag-tag features (indicated by prefix T), and suffix features (indicated by prefix M). We assume access to a method for extracting the suffix *-thy* from *slithy*, *-es* from *toves*, and ∅ from *the*, indicating that this word has no suffix. The resulting feature vector is,

$$\begin{aligned}\boldsymbol{f}(\text{the slithy toves, D J N}) =& \{\langle W:\text{the}, \text{D}\rangle, \langle M:\varnothing,\text{D}\rangle, \langle T:\lozenge,\text{D}\rangle\\ & \langle W:\text{slithy},\text{J}\rangle, \langle M:\text{-thy},\text{J}\rangle, \langle T:\text{D},\text{J}\rangle\\ & \langle W:\text{toves},\text{N}\rangle, \langle M:\text{-es},\text{N}\rangle, \langle T:\text{J},\text{N}\rangle\\ & \langle T:\text{N},\blacklozenge\rangle\}.\end{aligned}$$

We now consider several discriminative methods for learning feature weights in sequence labeling. In chapter 2, we considered three types of discriminative classifiers: perceptron, support vector machine, and logistic regression. Each of these classifiers has a structured equivalent, enabling it to be trained from labeled sequences rather than individual tokens.

Structured perceptron

The perceptron classifier updates its weights by increasing the weights for features that are associated with the correct label, and decreasing the weights for features that are associated with incorrectly predicted labels:

$$\hat{y} = \underset{y \in \mathcal{Y}}{\operatorname{argmax}} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}, y) \tag{6.66}$$

$$\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} + \boldsymbol{f}(\boldsymbol{x}, y) - \boldsymbol{f}(\boldsymbol{x}, \hat{y}). \tag{6.67}$$

We can apply exactly the same update in the case of structure prediction,

$$\hat{\boldsymbol{y}} = \underset{\boldsymbol{y} \in \mathcal{Y}(\boldsymbol{w})}{\operatorname{argmax}} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}) \tag{6.68}$$

$$\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} + \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}) - \boldsymbol{f}(\boldsymbol{w}, \hat{\boldsymbol{y}}). \tag{6.69}$$

This learning algorithm is called **structured perceptron**, because it learns to predict the structured output $\boldsymbol{y}$. The key difference is that instead of computing $\hat{\boldsymbol{y}}$ by enumerating the entire set $\mathcal{Y}$, we use the Viterbi algorithm to search this set efficiently. In this case, the output structure is the sequence of tags $(y_1, y_2, \ldots, y_M)$; the algorithm can be applied to other structured outputs as long as efficient inference is possible. As in perceptron classification, weight averaging is crucial to get good performance (see section 2.1).

Example For the example *They can fish*, suppose the reference tag sequence is N V V, but our tagger incorrectly returns the tag sequence N V N. Given **feature templates** $\langle w_m, y_m \rangle$ and $\langle y_{m-1}, y_m \rangle$, the corresponding structured perceptron update is:

$$\theta_{\langle \textit{fish}, \text{V} \rangle} \leftarrow \theta_{\langle \textit{fish}, \text{V} \rangle} + 1 \tag{6.70}$$

$$\theta_{\langle \textit{fish}, \text{N} \rangle} \leftarrow \theta_{\langle \textit{fish}, \text{N} \rangle} - 1 \tag{6.71}$$

$$\theta_{\langle \text{V}, \text{V} \rangle} \leftarrow \theta_{\langle \text{V}, \text{V} \rangle} + 1 \tag{6.72}$$

$$\theta_{\langle \text{V}, \text{N} \rangle} \leftarrow \theta_{\langle \text{V}, \text{N} \rangle} - 1 \tag{6.73}$$

$$\theta_{\langle \text{V}, \blacklozenge \rangle} \leftarrow \theta_{\langle \text{V}, \blacklozenge \rangle} + 1 \tag{6.74}$$

$$\theta_{\langle \text{N}, \blacklozenge \rangle} \leftarrow \theta_{\langle \text{N}, \blacklozenge \rangle} - 1. \tag{6.75}$$

Structured Support Vector Machines

Large-margin classifiers such as the support vector machine improve on the perceptron by learning weights that push the classification boundary away from the training instances. In many cases, large-margin classifiers outperform the perceptron, so we would like to apply similar ideas to sequence labeling. A support vector machine in which the output is a structured object, such as a sequence, is called a **structured support vector machine** (Tsochantaridis et al., 2004).[5]

In classification, we formalized the large-margin constraint as,

$$\forall y \neq y^{(i)}, \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}, y^{(i)}) - \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}, y) \geq 1, \tag{6.76}$$

which says that we require a margin of at least 1 between the scores for all labels y that are not equal to the correct label $y^{(i)}$. The weights $\boldsymbol{\theta}$ are then learned by constrained optimization (see section 2.2); for example, the PEGASOS algorithm (Shalev-Shwartz et al., 2007) applied stochastic subgradient descent to a Lagrangian expression that combines the margin constraints with a regularizer.

We can apply this idea to sequence labeling by formulating an equivalent set of constraints for all possible labelings $\mathcal{Y}(\boldsymbol{w})$ for an input $\boldsymbol{w}$. However, there are two problems with this idea. First, in sequence labeling, some predictions are more wrong than others: we may miss only one tag out of fifty, or we may get all fifty wrong. We would like our learning algorithm to be sensitive to this difference. Second, the number of contraints is equal to the number of possible labelings, which is exponentially large in the length of the sequence.

The first problem can be addressed by adjusting the constraint to require larger margins for more serious errors. Let $c(\boldsymbol{y}^{(i)}, \hat{\boldsymbol{y}}) \geq 0$ represent the **cost** of predicting label $\hat{\boldsymbol{y}}$ when the true label is $\boldsymbol{y}^{(i)}$. We can then generalize the margin constraint,

$$\forall \boldsymbol{y} \neq \boldsymbol{y}^{(i)}, \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)}) - \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}) \geq c(\boldsymbol{y}^{(i)}, \boldsymbol{y}). \tag{6.77}$$

This cost-augmented margin constraint specializes to the constraint in Equation 6.76 if we choose the delta function $c(\boldsymbol{y}^{(i)}, \boldsymbol{y}) = \delta(\boldsymbol{y}^{(i)} \neq \boldsymbol{y})$. For sequence labeling, we can instead use a structured cost function, such as the **Hamming cost**,

$$c(\boldsymbol{y}^{(i)}, \boldsymbol{y}) = \sum_{m=1}^{M} \delta(y_m^{(i)} \neq y_m). \tag{6.78}$$

With this cost function, we require that the true labeling be seperated from the alternatives by a margin that is proportional to the number of incorrect tags in each alternative labeling. Other cost functions are possible as well.

The second problem is that the number of constraints is exponential in the length of the sequence. This can be addressed by focusing on the prediction $\hat{\boldsymbol{y}}$

[5] This model is also known as a **max-margin Markov network** (Taskar et al., 2003), emphasizing that the scoring function is constructed from a sum of components, which are Markov independent.

that *maximally* violates the margin constraint. We find this prediction by solving the following **cost-augmented decoding** problem:

$$\hat{\boldsymbol{y}} = \underset{\boldsymbol{y} \neq \boldsymbol{y}^{(i)}}{\text{argmax}}\, \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}) - \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)}) + c(\boldsymbol{y}^{(i)}, \boldsymbol{y}) \tag{6.79}$$

$$= \max_{\boldsymbol{y} \neq \boldsymbol{y}^{(i)}} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}) + c(\boldsymbol{y}^{(i)}, \boldsymbol{y}), \tag{6.80}$$

where in the second line we drop the term $\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)})$, which is constant in $\boldsymbol{y}$.

We can now formulate the margin constraint for sequence labeling,

$$\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)}) - \max_{\boldsymbol{y} \in \mathcal{Y}(\boldsymbol{w})} \left(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}) + c(\boldsymbol{y}^{(i)}, \boldsymbol{y}) \right) \geq 0. \tag{6.81}$$

If the score for $\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)})$ is greater than the cost-augmented score for all alternatives, then the constraint will be met. Therefore we can maximize over the entire set $\mathcal{Y}(\boldsymbol{w})$, meaning that we can apply Viterbi directly.[6]

The name "cost-augmented decoding" is due to the fact that the objective includes the standard decoding problem, $\max_{\hat{\boldsymbol{y}}} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, \hat{\boldsymbol{y}})$, plus an additional term for the cost. Essentially, we want to train against predictions that are strong and wrong: they should score highly according to the model, yet incur a large loss with respect to the ground truth. We can then adjust the weights to reduce the score of these predictions.

For cost-augmented decoding to be tractable, the cost function must decompose into local parts, just as the feature function $\boldsymbol{f}(\cdot)$ does. The Hamming cost, defined above, obeys this property. To solve this cost-augmented decoding problem using the Hamming cost, we can simply add features $f_m(y_m) = \delta(y_m \neq y_m^{(i)})$, and assign a weight of 1 to these features. Decoding can then be performed using the Viterbi algorithm.

Are there cost functions that do not decompose into local parts? Suppose we want to assign a constant loss to any prediction $\hat{\boldsymbol{y}}$ in which k or more predicted tags are incorrect, and zero loss otherwise. This loss function is combinatorial over the predictions, and thus we cannot decompose it into parts.

As with support vector machine classifiers, we can formulate the learning problem as an unconstrained *primal form*, which combines a regularization term

[6] To maximize over the set $\mathcal{Y}(\boldsymbol{w}) \setminus \boldsymbol{y}^{(i)}$ we would need to an alternative version of Viterbi that returns the k-best predictions. K-best Viterbi may be useful for other reasons — for example, in interactive applications, it can be helpful to show the user multiple possible taggings. The design of k-best Viterbi is left an exercise.

on the weights and a Lagrangian for the constraints:

$$\min_{\boldsymbol{\theta}} \quad \frac{1}{2}||\boldsymbol{\theta}||_2^2 - C\left(\sum_i \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)}) - \max_{\hat{\boldsymbol{y}} \in \mathcal{Y}(\boldsymbol{w}^{(i)})} \left[\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}^{(i)}, \hat{\boldsymbol{y}}) + c(\boldsymbol{y}^{(i)}, \hat{\boldsymbol{y}})\right]\right), \tag{6.82}$$

In this formulation, C is a parameter that controls the tradeoff between the regularization term and the margin constraints. A number of optimization algorithms have been proposed for structured support vector machines, some of which are discussed in section 2.2. An empirical comparison by Kummerfeld et al. (2015) shows that stochastic subgradient descent — which is relatively easy to implement — is highly competitive, especially on the sequence labeling task of named entity recognition.

Conditional random fields

Structured perceptron is easy to implement, and structured support vector machines give excellent performance. However, sometimes we need to compute probabilities over labelings, $\mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w})$, and we would like to do this in a discriminative way. The **Conditional Random Field** (CRF; Lafferty et al., 2001) is a conditional probabilistic model for sequence labeling; just as structured perceptron is built on the perceptron classifier, conditional random fields are built on the logistic regression classifier.[7] The basic probability model is,

$$\mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w}) = \frac{\exp(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}))}{\sum_{\boldsymbol{y}' \in \mathcal{Y}(\boldsymbol{w})} \exp(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}'))}. \tag{6.83}$$

This is almost identical to logistic regression, but because the label space is now tag sequences, we require efficient algorithms for both **decoding** (searching for the best tag sequence given a sequence of words $\boldsymbol{w}$ and a model $\boldsymbol{\theta}$) and for **normalizing** (summing over all tag sequences). These algorithms will be based on the usual locality assumption on the feature function, $\boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}) = \sum_{m=1}^{M} \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m)$.

[7] The name "Conditional Random Field" is derived from **Markov random fields**, a general class of models in which the probability of a configuration of variables is proportional to a product of scores across pairs (or more generally, cliques) of variables in a **factor graph**. In sequence labeling, the pairs of variables include all adjacent tags $\langle y_m, y_{m-1} \rangle$. The probability is **conditioned** on the words $\boldsymbol{w}_{1:M}$, which are always observed, motivating the term "conditional" in the name.

Decoding in CRFs

Decoding — finding the tag sequence $\hat{\boldsymbol{y}}$ that maximizes $\mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w})$ — is a direct application of the Viterbi algorithm. The key observation is that the decoding problem does not depend on the denominator of $\mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w})$,

$$\begin{aligned}\hat{\boldsymbol{y}} =& \operatorname*{argmax}_{\boldsymbol{y}} \mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w})\\ =& \operatorname*{argmax}_{\boldsymbol{y}} \log \mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w})\\ =& \operatorname*{argmax}_{\boldsymbol{y}} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{y}, \boldsymbol{w}) - \log \sum_{\boldsymbol{y}' \in \mathcal{Y}(\boldsymbol{w})} e^{\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{y}', \boldsymbol{w})}\\ =& \operatorname*{argmax}_{\boldsymbol{y}} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{y}, \boldsymbol{w}).\end{aligned}$$

This is identical to the decoding problem for structured perceptron, so the same Viterbi recurrence as defined in Equation 6.26 can be used.

Learning in CRFs

As with logistic regression, we learn the weights $\boldsymbol{\theta}$ by minimizing the regularized negative log conditional probability,

$$\begin{aligned}\ell =& \frac{\lambda}{2}||\boldsymbol{\theta}||^2 - \sum_{i=1}^{N} \log \mathrm{p}(\boldsymbol{y}^{(i)} \mid \boldsymbol{w}^{(i)}; \boldsymbol{\theta}) && (6.84)\\ =& \frac{\lambda}{2}||\boldsymbol{\theta}||^2 - \sum_{i=1}^{N} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)}) + \log \sum_{\boldsymbol{y}' \in \mathcal{Y}(\boldsymbol{w}^{(i)})} \exp\left(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}')\right), && (6.85)\end{aligned}$$

where λ controls the amount of regularization. We will optimize $\boldsymbol{\theta}$ by moving along the gradient of this loss. Probabilistic programming environments, such as THEANO (Bergstra et al., 2010) and TORCH (Collobert et al., 2011), can compute this gradient using automatic differentiation. However, it is worth deriving the gradient to understand how this model works, and why learning is computationally tractable.

As in logistic regression, the gradient includes a difference between observed and expected feature counts:

$$\frac{d\ell}{d\theta_j} = \lambda\theta_j + \sum_{i=1}^{N} E[f_j(\boldsymbol{w}^{(i)}, \boldsymbol{y})] - f_j(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)}), \tag{6.86}$$

where $f_j(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)})$ refers to the count of feature j for token sequence $\boldsymbol{w}^{(i)}$ and tag sequence $\boldsymbol{y}^{(i)}$.

The expected feature counts are computed by summing over all possible labelings of the word sequence,

$$E[f_j(\boldsymbol{w}^{(i)}, \boldsymbol{y})] = \sum_{\boldsymbol{y} \in \mathcal{Y}(\boldsymbol{w}^{(i)})} \mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w}^{(i)}; \boldsymbol{\theta}) f_j(\boldsymbol{w}^{(i)}, \boldsymbol{y}) \tag{6.87}$$

This looks bad: it is a sum over an exponential number of labelings. To solve this problem, we again rely on the assumption that the overall feature vector decomposes into a sum of local feature vectors, which we exploit to compute the expected feature counts as a sum across the sequence:

$$\begin{align}
E[f_j(\boldsymbol{w}, \boldsymbol{y})] = & \sum_{\boldsymbol{y} \in \mathcal{Y}(\boldsymbol{w})} \mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w}; \boldsymbol{\theta}) f_j(\boldsymbol{w}, \boldsymbol{y}) \tag{6.88} \\
= & \sum_{\boldsymbol{y} \in \mathcal{Y}(\boldsymbol{w})} \mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w}; \boldsymbol{\theta}) \sum_{m=1}^{M} f_j(\boldsymbol{w}, y_m, y_{m-1}, m) \tag{6.89} \\
= & \sum_{m=1}^{M} \sum_{\boldsymbol{y} \in \mathcal{Y}(\boldsymbol{w})} \mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w}; \boldsymbol{\theta}) f_j(\boldsymbol{w}, y_m, y_{m-1}, m) \tag{6.90} \\
= & \sum_{m=1}^{M} \sum_{k,k'}^{\mathcal{Y}} \sum_{\boldsymbol{y}: y_{m-1}=k', y_m=k} \mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w}; \boldsymbol{\theta}) f_j(\boldsymbol{w}, k, k', m) \tag{6.91} \\
= & \sum_{m=1}^{M} \sum_{k,k'}^{\mathcal{Y}} f_j(\boldsymbol{w}, k, k', m) \sum_{\boldsymbol{y}: y_{m-1}=k', y_m=k} \mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w}; \boldsymbol{\theta}) \tag{6.92} \\
= & \sum_{m=1}^{M} \sum_{k,k'}^{\mathcal{Y}} f_j(\boldsymbol{w}, k', k, m) \Pr(Y_{m-1} = k', Y_m = k \mid \boldsymbol{w}; \boldsymbol{\theta}). \tag{6.93}
\end{align}$$

This derivation works by interchanging the sum over sequences with the sum over m. At each position in the sequence, we need only the marginal probability of the tag bigram, $\Pr(Y_{m-1} = k', Y_m = k \mid \boldsymbol{w}; \boldsymbol{\theta})$. These tag bigram marginals are also used in unsupervised approaches to sequence labeling. In principle, these marginals still require a sum over the exponentially many label sequences in which $Y_{m-1} = k'$ and $Y_m = k$. However, they can be computed efficiently using the **forward-backward algorithm**.

*Forward-backward algorithm

Recall that we previously used the Forward algorithm to compute marginal probabilities $\mathrm{p}(y_m, \boldsymbol{w}_{1:m})$ in the hidden Markov model. We now derive a more general version of the algorithm, in which label sequences are scored in terms of **potentials** $\psi_m(k, k')$:

$$\psi_m(k, k') \triangleq \exp(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, k, k', m)) \tag{6.94}$$

$$\mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w}) = \frac{\prod_{m=1}^{M} \psi_m(y_m, y_{m-1})}{\sum_{\boldsymbol{y}'} \prod_{m=1}^{M} \psi_m(y'_m, y'_{m-1})}. \tag{6.95}$$

It should be clear that Equation 6.95 simply expresses the CRF conditional likelihood, under a change of notation.

Using this notation, we can compute the tag bigram marginals as,

$$\Pr(Y_{m-1} = k', Y_m = k \mid \boldsymbol{w}; \boldsymbol{\theta}) = \frac{\sum_{\boldsymbol{y}:y_m=k, y_{m-1}=k'} \prod_{n=1}^{M} \psi_n(y_n, y_{n-1})}{\sum_{\boldsymbol{y}'} \prod_{n=1}^{M} \psi_n(y'_n, y'_{n-1})}. \tag{6.96}$$

where the denominator is the marginal $\mathrm{p}(\boldsymbol{w}_{1:M}) = \sum_{\boldsymbol{y}} \mathrm{p}(\boldsymbol{w}, \boldsymbol{y}_{1:M})$, sometimes known as the **partition function**.[8] Let us now consider how to compute each of these terms efficiently.

Computing the numerator In Equation 6.96, we sum over all tag sequences that include the transition $(Y_{m-1} = k') \rightarrow (Y_m = k)$. Because we are only interested in sequences that include this arc, we can decompose this sum into three parts: the sum over **prefixes** $\boldsymbol{y}_{1:m-1}$, the transition $(Y_{m-1} = k') \rightarrow (Y_m = k)$, and the sum over **suffixes** $\boldsymbol{y}_{m:M}$,

$$\begin{aligned}\sum_{\boldsymbol{y}:Y_m=k, Y_{m-1}=k'} \prod_{n=1}^{M} \psi_n(y_n, y_{n-1}) = & \sum_{\boldsymbol{y}_{1:m-1}:y_{m-1}=k'} \prod_{n=1}^{m-1} \psi_n(y_n, y_n - 1) \\ & \times \psi_m(k, k') \\ & \times \sum_{\boldsymbol{y}_{m:M}:y_m=k} \prod_{n=m+1}^{M} \psi_n(y_n, y_{n-1}).\end{aligned} \tag{6.97}$$

[8] The terminology of "potentials" and "partition functions" is due to a connection with statistical mechanics (Bishop, 2006).

The result is product of three terms: a score for getting to the position $(Y_{m-1} = k')$, a score for the transition from k' to k, and a score for finishing the sequence from $(Y_m = k)$. Let us define the first term as a **forward variable**,

$$\begin{aligned}
\alpha_m(k) \triangleq & \sum_{\boldsymbol{y}_{1:m}:y_m=k} \prod_{n=1}^{m} \psi_n(y_n, y_{n-1}) && (6.98)\\
= & \sum_{k'} \psi_m(k,k') \sum_{\boldsymbol{y}_{1:m-1}:y_{m-1}=k'} \prod_{n=1}^{m-1} \psi_n(y_n, y_{n-1}) && (6.99)\\
= & \sum_{k'} \psi_m(k,k')\alpha_{m-1}(k'). && (6.100)
\end{aligned}$$

Thus, we compute the forward variables while moving from left to right over the trellis. This forward recurrence is a generalization of the forward recurrence defined in section 6.4. If we set $\psi_m(k,k') = \text{p}_E(w_m \mid Y_m = k) \Pr(Y_m = k \mid Y_{m-1} = k')$, we exactly recover the HMM forward variable $\alpha_m(k) = \text{p}(\boldsymbol{w}_{1:m}, Y_m = k)$.

The third term of Equation 6.97 can also be defined recursively, this time moving over the trellis from right to left. The resulting recurrence is called the **backward algorithm**:

$$\begin{aligned}
\beta_{m-1}(k) \triangleq & \sum_{\boldsymbol{y}_{m-1:M}:y_{m-1}=k} \prod_{n=m}^{M} \psi_n(y_n, y_{n-1}) && (6.101)\\
= & \sum_{k'} \psi_m(k',k) \sum_{\boldsymbol{y}_{m:M}:y_m=k'} \prod_{n=m+1}^{M} \psi_n(y_n, y_{n-1}) && (6.102)\\
= & \sum_{k'} \psi_m(k',k)\beta_m(k'). && (6.103)
\end{aligned}$$

In practice, numerical stability requires that we use log-potentials rather than potentials, $\log \psi_m(y_m, y_{m-1}) = \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m)$. Then the sums must be replaced with log-sum-exp:

$$\begin{aligned}
\log \alpha_m(k) = & \log \sum_{k'} \exp\left(\log \psi_m(k,k') + \log \alpha_{m-1}(k')\right) && (6.104)\\
\log \beta_{m-1}(k) = & \log \sum_{k'} \exp\left(\log \psi_m(k',k) + \log \beta_m(k'))\right). && (6.105)
\end{aligned}$$

Both the forward and backward algorithm operate on the trellis, which implies a space complexity $\mathcal{O}(MK)$. Because they require computing a sum over K terms at each node in the trellis, their time complexity is $\mathcal{O}(MK^2)$.

Computing the normalization term The normalization term, sometimes abbreviated as Z, can be written as,

$$
\begin{aligned}
Z \triangleq &\sum_{\boldsymbol{y}} \mathrm{p}(\boldsymbol{w}, \boldsymbol{y}) && (6.106)\\
=&\sum_{\boldsymbol{y}} \exp\left(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y})\right) && (6.107)\\
=&\sum_{\boldsymbol{y}} \prod_{m=1}^{M} \exp\left(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m)\right) && (6.108)\\
=&\sum_{\boldsymbol{y}} \prod_{m=1}^{M} \psi_m(y_m, y_{m-1}). && (6.109)
\end{aligned}
$$

This term can be computed directly from either the forward or backward probabilities:

$$
\begin{aligned}
Z =&\sum_{\boldsymbol{y}} \prod_{m=1}^{M} \psi_m(y_m, y_{m-1}) && (6.110)\\
=&\alpha_{M+1}(\blacklozenge) && (6.111)\\
=&\beta_0(\lozenge). && (6.112)
\end{aligned}
$$

CRF learning: wrapup Having computed the forward and backward variables, we can compute the desired marginal probability as,

$$
P(Y_{m-1} = k', Y_m = k \mid \boldsymbol{w}_{1:M}) = \frac{\alpha_{m-1}(k')\psi_m(k, k')\beta_m(k)}{Z}. \qquad (6.113)
$$

This computation is known as the **forward-backward algorithm.** From the resulting marginals, we can compute the feature expectations $E[f_j(\boldsymbol{w}, \boldsymbol{y})]$; from these expectations, we compute a gradient on the weights $\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$. Stochastic gradient descent or quasi-Newton optimization can then be applied. As the optimization algorithm changes the weights, the potentials change, and therefore so do the marginals. Each iteration of the optimization algorithm therefore requires recomputing the forward and backward variables for each training instance. [9]

[9] The `CRFsuite` package implements several learning algorithms for CRFs (`http://www.chokkan.org/software/crfsuite/`).

6.6 *Unsupervised sequence labeling

In unsupervised sequence labeling, we want to induce a Hidden Markov Model from a corpus of unannotated text $\boldsymbol{w}^{(1)}, \boldsymbol{w}^{(2)}, \ldots, \boldsymbol{w}^{(N)}$. This is an example of the general problem of **structure induction**, which is the unsupervised version of **structure prediction**. The tags that result from unsupervised sequence labeling might be useful for some downstream task, or they might help us to better understand the language's inherent structure.

Unsupervised learning in hidden Markov models can be performed using the **Baum-Welch algorithm**, which combines forward-backward with expectation-maximization (EM). In the M-step, we compute the HMM parameters from expected counts:

$$\Pr(W = i \mid Y = k) = \phi_{k,i} = \frac{E[\text{count}(W = i, Y = k)]}{E[\text{count}(Y = k)]}$$

$$\Pr(Y_m = k \mid Y_{m-1} = k') = \lambda_{k',k} = \frac{E[\text{count}(Y_m = k, Y_{m-1} = k')]}{E[\text{count}(Y_{m-1} = k')]}$$

The expected counts are computed in the E-step, using the forward and backward variables as defined in Equation 6.100 and Equation 6.103. Because we are working in a hidden Markov model, we define the potentials as,

$$\psi_m(k, k') = \mathrm{p}_E(w_m \mid Y_m = k; \boldsymbol{\phi}) \Pr(Y_m = k \mid Y_{m-1} = k'; \lambda). \tag{6.114}$$

The expected counts are then,

$$E[\text{count}(W = i, Y = k)] = \sum_m \Pr(Y_m = k \mid \boldsymbol{w}_{1:M}) \delta(W_m = i) \tag{6.115}$$

$$= \sum_m \frac{\Pr(Y_m = k, \boldsymbol{w}_{1:m}) \mathrm{p}(\boldsymbol{w}_{m+1:M} \mid Y_m = k)}{\mathrm{p}(\boldsymbol{w}_{1:M})} \delta(w_m = i) \tag{6.116}$$

$$= \frac{1}{\alpha_M(\blacklozenge)} \sum_m \alpha_m(k) \beta_m(k) \delta(w_m = i) \tag{6.117}$$

We use the chain rule to separate $\boldsymbol{w}_{1:m}$ and $\boldsymbol{w}_{m+1:M}$, and then use the definitions of the forward and backward variables. In the final step, we normalize by

$\mathrm{p}(\boldsymbol{w}_{1:M}) = \alpha_{M+1}(\blacklozenge) = \beta_0(\Diamond)$:

$$
\begin{aligned}
E[\text{count}(Y_m = k, Y_{m-1} = k')] =& \sum_m P(Y_m = k, Y_{m-1} = k' \mid \boldsymbol{w}_{1:M}) && (6.118)\\
\propto& \sum_m P(Y_{m-1} = k', \boldsymbol{w}_{1:m-1}) P(w_{m+1:M} \mid Y_m = k)\\
& \times P(w_m, Y_m = k \mid Y_{m-1} = k') && (6.119)\\
=& \sum_m P(Y_{m-1} = k', \boldsymbol{w}_{1:m-1}) P(w_{m+1:M} \mid Y_m = k)\\
& \times \mathrm{p}(w_m \mid Y_m = k) P(Y_m = k \mid Y_{m-1} = k') && (6.120)\\
=& \sum_m \alpha_{m-1}(k')\beta_m(k)\phi_{k,w_m}\lambda_{k' \to k}. && (6.121)
\end{aligned}
$$

Again, we use the chain rule to separate out $\boldsymbol{w}_{1:m-1}$ and $\boldsymbol{w}_{m+1:M}$, and use the definitions of the forward and backward variables. The final computation also includes the parameters ϕ and λ, which govern (respectively) the emission and transition properties between $w_m, y_m,$ and y_{m-1}. Note that the derivation only shows how to compute this to a constant of proportionality; we would divide by $\mathrm{p}(\boldsymbol{w}_{1:M})$ to go from the joint probability $P(Y_{m-1} = k', Y_m = k, \boldsymbol{w}_{1:M})$ to the desired conditional $\Pr(Y_{m-1} = k', Y_m = k \mid \boldsymbol{w}_{1:M})$.

Linear dynamical systems

The forward-backward algorithm can be viewed as Bayesian state estimation in a discrete state space. In a continuous state space, $y_m \in \mathbb{R}$, the equivalent algorithm is the **Kalman Smoother**. It also computes marginals $\mathrm{p}(y_m \mid \boldsymbol{x}_{1:M})$, using a similar two-step algorithm of forward and backward passes. Instead of computing a trellis of values at each step, we would compute a probability density function $q_{y_m}(y_m; \mu_m, \Sigma_m)$, characterized by a mean μ_m and a covariance Σ_m around the latent state. Connections between the Kalman Smoother and the forward-backward algorithm are elucidated by Minka (1999) and Murphy (2012).

Alternative unsupervised learning methods

As noted in section 4.4, expectation-maximization is just one of many techniques for structure induction. One alternative is to use a family of randomized algorithms called **Markov Chain Monte Carlo (MCMC)**. In these algorithms, we compute a marginal distribution over the latent variable $\boldsymbol{y}$ **empirically**, by drawing random samples. The randomness explains the "Monte Carlo" part of the name; typically, we employ a Markov Chain sampling procedure, meaning that each sample is drawn from a distribution that depends only on the previous sample (and not on the entire sampling history). A simple MCMC algorithm is **Gibbs Sampling**, in which we iteratively sample each y_m conditioned on all the others (Finkel et al., 2005):

$$\mathrm{p}(y_m \mid \boldsymbol{y}_{-m}, \boldsymbol{w}_{1:M}) \propto \mathrm{p}(w_m \mid y_m)\mathrm{p}(y_m \mid \boldsymbol{y}_{-m}). \tag{6.122}$$

Gibbs Sampling has been applied to unsupervised part-of-speech tagging by Goldwater and Griffiths (2007). *Beam sampling* is a more sophisticated sampling algorithm, which randomly draws entire sequences $\boldsymbol{y}_{1:M}$, rather than individual tags y_m; this algorithm was applied to unsupervised part-of-speech tagging by Van Gael et al. (2009).

EM is guaranteed to find only a local optimum; MCMC algorithms will converge to the true posterior distribution $\mathrm{p}(\boldsymbol{y}_{1:M} \mid \boldsymbol{w}_{1:M})$, but this is only guaranteed in the limit of infinite samples. Recent work has explored the use of **spectral learning** for latent variable models, which use matrix and tensor decompositions to provide guaranteed convergence under mild assumptions (Song et al., 2010; Hsu et al., 2012).

Chapter 7

Applications of sequence labeling

- Part-of-speech tagging: *Go*/V *to*/P *Georgia*/N *Tech*/N *next*/J *year*/N ./.
- Named entity recognition:[1] *Go*/O *to*/O *Georgia*/B-ORG *Tech*/I-ORG *next*/B-DATE *year*/I-DATE ./O
- Phrase chunking: *Go*/B-VP *to*/B-PP *Georgia*/B-NP *Tech*/I-NP *next*/B-NP *year*/I-NP ./O

7.1 Part-of-speech tagging

Words can be grouped into rough classes based on syntax.

- Why is *colorless green ideas sleep furiously* more acceptable than *ideas colorless furiously green sleep*?
- Why is *teacher strikes idle children* ambiguous?

In both examples, word classes can provide an explanation.

- Word classes have strong ordering constraints:
  - J J N V R is relatively likely. This is the tag sequence for *colorless green ideas sleep furiously*. The abbreviation J means adjective, N means noun, V means verb, and R means adverb.

[1] These examples show **BIO** notation, in which spans such as ORG (organization) or NP (noun phrase) are delimited using prefixes B- and I-. The prefixes indicate whether each token is at the **b**eginning or **i**nside of the span; the tag O is reserved for tokens that are outside any span. For now, we will just think of B-ORG, I-ORG, O, etc, as separate tags; see chapter 18 for more.

  - N J R J V is very unlikely in English. Do you see why?

- Ambiguity about word class leads to very different interpretations:

  (7.1) *teacher/N strikes/N idle/V children/N*

  (7.2) *teacher/N strikes/V idle/J children/N* (ouch!)

So clearly we have intuitions about a few parts-of-speech already: noun, verb, adjective, adverb. Jurafsky and Martin (2009) describe these as the four major **open** word classes, although apparently [todo: ?] not all languages have all of them.

What other parts of speech are there?

- The Penn Treebank defines a set of 45 POS tags for English.[2]
- The Brown corpus defines a set of 87 POS tags for English.[3]
- Petrov et al. (2012) define a "universal" set of 12 tags, which are supposed to apply across many languages.

To understand the linguistic differences between these tagsets, let's look at an example:

(7.3) My name is Ozymandias, king of kings:
Look on my works, ye Mighty, and despair!

The part-of-speech tags for this couplet from Ozymandias are shown in Table 7.1.

Tagset granularity

All tagsets distinguish basic categories like nouns, pronouns, verbs, adjectives, and punctuation. The Brown tagset includes a number of fine-grained distinctions:

- specific tags for the *be*, *do*, and *have* verbs, which the other two tagsets just lump in with other verbs;
- distinct tags for possessive determiners (*my name*) and possessive pronouns (*mine*);

[2] `http://www.comp.leeds.ac.uk/ccalas/tagsets/upenn.html`

[3] `http://www.comp.leeds.ac.uk/ccalas/tagsets/brown.html`

	Brown	PTB	Universal
My	possessive determiner (DD$)	possessive pronoun (PRP$)	pronoun (PRON)
name	noun, singular, common (NN)	NN	NOUN
is	verb "to be" 3rd person, singular (BEZ)	verb 3rd person, singular (VBZ)	VERB
Ozymandias	proper noun, singular (NP)	proper noun, singular (NNP)	NOUN
,	comma (,)	comma (,)	punctuation (.)
king	NN	NN	NOUN
of	preposition (IN)	preposition (IN)	adposition (ADP)
kings	noun, plural, common (NNS)	NNS	NOUN
:	colon (:)	mid-sentence punc (:)	.
Look	verb, base: uninflected present, imperative, or infinite (VB)	VB	VERB
on	IN	IN	ADP
my	DD$	PRP$	PRON
works	NNS	NNS	NOUN
ye	personal pronoun, nominative, non 3S (PPSS)	personal pronoun, nominative (PRP)	PRON
mighty	adjective (JJ)	JJ	adjective (ADJ)
,	comma (,)	comma (,)	punctuation (.)
and	coordinating conjunction (CC)	CC	conjunction (CONJ)
despair	VB	VB	VERB

Table 7.1: Part-of-speech annotations from three tagsets for the first couple of the poem Ozymandias.

- distinct tags for the third-person singular pronouns (e.g., *it*, *he*) and other pronouns (e.g., *they*, *we*, *I*).

In contrast, the Universal tagset aggressively groups categories that are distinguished in the other tagsets:

- all nouns are grouped, ignoring number and the proper/common distinction (see below);
- all verbs are grouped, ignoring inflection;
- preposition and postpositions are grouped as "adpositions";
- all punctuation is grouped;
- coordinating and subordinating conjunctions (e.g. *and* versus *that*) are grouped.

The Penn Treebank strikes a middle ground between these two relative extremes. But which is right? It depends. The Brown tags can be useful for certain applications, and they may have strong tag-to-tag relations that make tagging easier, as described in the next chapter). But they are more expensive to annotate. The Universal tags are intended to generalize across many languages and many types of text, and should be easier to annotate.

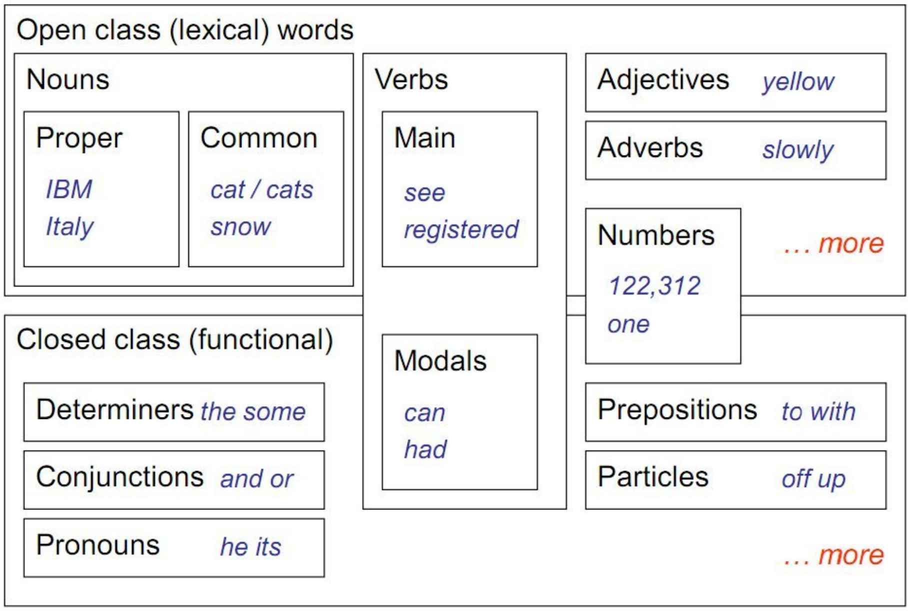


Figure 7.1: [todo: attribution?]

Linguistics of parts of speech

As usual, Bender (2013) provides a useful linguistic perspective.

- **Nouns** describe entities and concepts
  - **Proper nouns** name specific people and entities: *Georgia Tech*, *Janet*, *Buddhism*. In English, proper nouns are usually capitalized. The Penn Treebank (PTB) tags are: NNP (singular), NNPS (plural).
  - **Common nouns** cover all other nouns. In English, they are often preceded by determiners, e.g. *the book*, *a university*, *some people*. Common nouns decompose into two main types:
    - **Count nouns** have a plural and need an article in the singular, *dogs*, *the dog*;
    - **Mass nouns** don't have a plural and don't need an article in the singular:

      (7.4) *snow is cold*

      (7.5) *gas is expensive*
  - **Pronouns** refer to specific noun phrases or entities or events.
    - **Personal pronouns** refer to people or entities: *you, she, I, it, me*. The PTB tag is PRP.
    - **Possessive pronouns** are pronouns that indicate possession: *your, her, my, its, one's, our*. The PTB tag is PRP$.
    - **Wh-pronouns** (WP) are used in question forms, and as relative pronouns:

      (7.6) *Where are you going?*

      (7.7) *The man who wasn't there.*

    Unlike other nouns, the set of possible pronouns cannot be expanded. It is a **closed class**. Can you think of other closed class word groups?

- **Verbs** describe activities, processes, and events. For example, *eat, write, sleep* are all verbs.
  - The Penn Treebank differentiates verbs by morphology: VB (infinitive), VBD (past), VBG (present participle), VBN (past participle), VBZ (present 3rd person singular), VBP (present, non-3rd person singular).

- **Modals** are a closed subclasses of verbs, such as (*should, can, will, must*). They get PTB tag MD.
- The verb *to be* requires special treatment, as it must appear with a predicative adjective or noun, e.g.

  (7.8) *She is hungry.*

  (7.9) *We are Georgians.*

  The verbs *is* and *are* in these cases are called **copula**. The Brown Tagset distinguishes copula, but the PTB does not. More generally, in **light verb** constructions, the meaning is largely shaped by a predicative adjective, e.g. *he got fired*, [todo: more examples].
- **Auxiliary** verbs include *be, have, will*, which form complex tenses in English, e.g. *we will have done it twice*. Recall from chapter 9 that English makes extensive use of auxiliary verbs to determine the tense, while other languages, such as French, rely more on morphology.
  - Another auxiliary verb is *do*, as used in questions and negation, e.g.

    (7.10) *Did you eat yet?*

    (7.11) *We did not take your bagels.*
  - The Brown corpus has special tags for HAVE and DO, but the PTB does not.

- **Adjectives** describe properties of entities: in the Ozymandias eaxmple, the adjectives include *antique, vast, trunkless*. In English, adjectives can be used in two ways:

  - **Attributive**: *an antique land;*
  - **Predicative**: *the land was antique.*

  Adjectives may be **gradable**, meaning that they have a **comparative form** (e.g., *bigger*, *smellier*) **superlative form** (*biggest*, *smelliest*). Adjectives like *antique* are not gradable.

  - With *big*, we can move to comparative form by adding the suffix *-est*. This is an example of agglutinative morphology, since the comparative morpheme is added to the stem as an affix. But there are adjectives in English where the relationship between the base and comparative forms is not agglutinative, but fusional. One example *good*, *better*, *best*; can you think of any others?

  - The PTB distinguishes these forms with three tags: JJ, JJR, JJS.

- **Adverbs** describe properties of events.
  - **Manner**: *slowly, slower, fast, hesitantly*
  - **Degree**: *extremely, very, highly*
  - Adverbs may be directional or locative. In the following examples, the bolded words are all adverbs.

    (7.12) *She lives **downstairs**.*

    (7.13) *I study **here**.*

    (7.14) *Go **left** at the first traffic light.*

  - Adjectives also include temporal information, such as *yesterday, Monday*, and *soon*.
  - Besides verbs, adverbs may also modify sentences, adjectives, or other adverbs.

    (7.15) ***Apparently**, the **very** ill man walks **extremely slowly**.*

    In this example, *very* modifies the adjective *ill*, *slowly* modifies the verb *walks*, *extremely* modifies the adverb *slowly*, and *apparently* modifies the entire sentence that follows it.
  - Like adjectives, adverbs may also be gradable. The PTB distinguishes graded adjectives with the tags RB, RBR, RBS.

- **Prepositions** are a closed class of words that can come before noun phrases, forming a prepositional phrase that relates the noun phrase to something else in the sentence.
  - *I eat sushi **with** soy sauce*. The prepositional phrase **attaches** to the noun *sushi*.
  - *I eat sushi **with** chopsticks*. The prepositional phrase here attaches to the verb *eat*.

  The preposition *To* gets its own tag TO, because it forms the **infinitive** with bare form verbs (VB), e.g. *I want to eat*. All other prepositions are tagged IN in the PTB.

- **Coordinating conjunctions** (PTB tag: CC) join two elements, (7.16) *vast* ***and*** *trunkless legs*

(7.17) *She plays backgammon* ***or*** *she does homework.*

(7.18) *She eats* ***and*** *drinks quickly.*

(7.19) *Sandeep lives north of Midtown* ***and*** *south of Buckhead.*

(7.20) *Max cooked,* ***and*** *Abigail ate, all the pizza.*

- **Subordinating conjunctions** introduce a subordinate clause, e.g.

  (7.21) *She thinks* ***that*** *Chomsky is wrong about language models.*

  The PTB tag here is IN.

- **Particles** are words that come with verbs and can change their meaning to a new **phrasal verb**, e.g.,

  (7.22) *Come* ***on***.

  (7.23) *He brushed himself* ***off***

  (7.24) *Let's check* ***out*** *that new restaurant*.

  Particles are a closed class, and are tagged RP in the PTB.

- **Determiners** (PTB tag: DT) are a closed class of words that precede noun phrases.
  - Articles: *the, an, a*
  - Demonstratives: *this, these, that*
  - Quantifiers: *some, every, few*
  - Wh-determiners: e.g., ***Which*** *bagel should I choose?*, *Do you know* ***when*** *it will be ready?*

- **Oddballs**
  - **Existential there**, e.g. *There is no way out of here*, gets its own tag, EX.
  - So does the possessive ending *'s*, which is POS. Recall that possessive pronouns don't have this ending, so they get a special tag, $PRP$$.
  - Other special tags are reserved for numbers (CD), list items (LS), commas (,), and other non-alphabetic symbols.

The	U.S.	Army	captured	Atlanta	on	May	14	,	1864	.
B-ORG	I-ORG	I-ORG	O	B-LOC	O	B-DATE	I-DATE	I-DATE	I-DATE	O

Table 7.2: BIO notation for named entity recognition

7.2 Shallow parsing

7.3 Named entity recognition

A standard approach is to tagging named entity spans is to use discriminative sequence labeling methods such as conditional random fields and structured perceptron. As described in chapter 6, these methods use the Viterbi algorithm to search over all possible label sequences, while scoring each sequence using a feature function that decomposes across adjacent tags. Named entity recognition is formulated as a tagging problem by assinging each word token to a tag from a tagset. However, there is a major difference from part-of-speech tagging: in NER we need to recover **spans** of tokens, such as *The United States Army*. To do this, the tagset must distinguish tokens that are at the **b**eginning of a span from tokens that are **i**nside a span.

BIO notation This is accomplished by the "BIO notation", shown in Table 18.1. Each token at the beginning of a name span is labeled with a B- prefix; each token within a name span is labeled with an I- prefix. Tokens that are not parts of name spans are labeled as O. From this representation, it is unambiguous to recover the entity name spans within a labeled text. Another advantage is from the perspective of learning: tokens at the beginning of name spans may have different properties than tokens within the name, and the learner can exploit this. This insight can be taken even further, with special labels for the **l**ast tokens of a name span, and for **u**nique tokens in name spans, such as *Atlanta* in the example in Table 18.1. This is called BILOU notation, and has been shown to yield improvements in supervised named entity recognition Ratinov and Roth (2009).[todo: check this cite]

Entity types The number of possible entity types depends on the labeled data. An early dataset was released as part of a shared task in the Conference on Natural Language Learning (CoNLL), containing entity types LOC (location), ORG (organization), and PER (person). Later work has distinguished additional entity types, such as dates, [todo: etc]. [todo: find cites] Special purpose corpora have been built for domains such as biomedical text, where entities include protein types [todo: etc].

Features The use of Viterbi decoding restricts the feature function $\boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y})$ to $\sum_m \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m)$, so that each feature can consider only local adjacent tags. Typical features include tag transitions, word features for w_m and its neighbors, character-level features for prefixes and suffixes, and “word shape” features to capture capitalization. As an example, base features for the word *Army* in the example in Table 18.1 include:

$$\begin{aligned}\langle&\textsc{curr-word}{:}\textit{Army},\\ &\textsc{prev-word}{:}\textit{U.S.},\\ &\textsc{next-word}{:}\textit{captured},\\ &\textsc{prefix-1}{:}\textit{A-},\\ &\textsc{prefix-2}{:}\textit{Ar-},\\ &\textsc{suffix-1}{:}\textit{-y},\\ &\textsc{suffix-2}{:}\textit{-my},\\ &\textsc{shape}{:}\textit{Xxxx}\rangle\end{aligned}$$

Another source of features is to use **gazzeteers**: lists of known entity names. For example, it is possible to obtain from the U.S. Social Security Administration a list of [todo: hundreds of thousands] of frequently used American names — more than could be observed in any reasonable annotated corpus. Tokens or spans that match an entry in a gazetteer can receive special features; this provides a way to incorporate hand-crafted resources such as name lists in a learning-driven framework.

Features in recent state-of-the-art systems are summarized in papers by **?** and Ratinov and Roth (2009).

7.4 Dialogue acts

Chapter 8

Finite-state automata

Consider the following problems:

- Segment a word into its stem and affixes: *impossibility* → *im+possibl+ity*.
- Convert a sequence of morphemes like *im+possible+ity* into the correct sequence of characters (*impossibility*).
- Decide whether a given word is morphotactically correct, or more generally, rank all the possible realizations for a morphological expression like NEGATION + *possible*: *impossible*, *inpossible*, *nonpossible*, *unpossible*, etc.
- Given a speech utterance and a large set of potential text transcriptions, choose the one with the highest probability according to an n-gram language model.
- Perform context-sensitive spelling correction, so as to correct examples like *their at piece* to *they're at peace*.

All of these problems relate to the content of the previous two chapters — language models and morphology — but none of them seem easily solved by supervised classifiers. This chapter presents a new tool for language technology: finite state automata. Finite-state automata are particularly suited for scoring strings (sequences of characters, words, morphemes, or phonemes), and for converting one string into another. A key advantage of finite state automata is their modularity: the output of one finite-state transducer can be the input for another, allowing the combination of simple components into cascades with rich and complex behaviors.

Finite-state automata are a formalism for representing a subset of formal languages, the **regular** languages; these are languages that can be defined with regular expressions. While there is strong evidence that natural language is not regular — that is, the question of whether a given sentence is grammatical cannot be answered with any regular expression — finite state automata can be used as the building block for a surprisingly wide range of applications in language technology.[1]

8.1 Automata and languages

Finite state automata emerge from formal language theory. Here are some basic formalisms that will be used throughout this chapter:

- An **alphabet** Σ is a set of symbols, e.g. $\{$*a*,*b*,*c*,. . . ,*z*$\}$, or $\{$*aardvark*, *abacus*, . . . , *zyxt*$\}$.
- A **string** ω is a sequence of symbols, $\omega \in \Sigma^*$. The empty string ϵ contains zero symbols.
- A **language** $L \subseteq \Sigma^*$ is a set of strings.
- An **automaton** is an abstract model of a computer, which reads a string $\omega \in \Sigma^*$, and determines whether or not $\omega \in L$.

This seems a very different notion of "language" than English or Hindi. But could we think of these natural languages in the same way as formal languages? If *impossible* is acceptable as an English word but *unpossible* is not, might it be possible to build an automaton that formalizes the underlying linguistic distinction?

Finite-state automata

A finite-state **acceptor** (FSA) is a special type of automaton, which is capable of modeling some, but not all languages. Formally, finite-state automata are defined by a tuple $M = \langle Q, \Sigma, q_0, F, \delta \rangle$, consisting of:

- a finite alphabet Σ of input symbols;
- a finite set of **states** $Q = \{q_0, q_1, \ldots, q_n\}$;

[1] A more formal treatment of finite state automata and their applications to language is offered by Mohri et al. (2002). Knight and May (2009) show how finite-state automata can be composed together to create impressive applications, focusing on **transliteration** of words and names between languages with different scripts. Here, we'll build the formalism from the ground up, starting with finite-state acceptors, then adding weights, and then adding transduction, finally arriving at the same sorts of applications.

- a **start state** $q_0 \in Q$;
- a set of **final states** $F \subseteq Q$;
- a **transition function** $\delta : Q \times \Sigma \to 2^Q$. The transition function maps from a state and an input symbol to a **set** of possible resulting states.

Given this definition, M accepts a string ω if there is a path from q_0 to any state $q_i \in F$ that consumes all of the symbols in ω. If M accepts ω, this means that ω is in the formal language L defined by M.

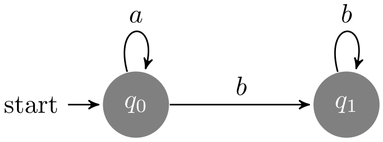
Example Consider the following FSA, M_1.

$$\Sigma =\{a, b\} \tag{8.1}$$

$$Q =\{q_0, q_1\} \tag{8.2}$$

$$F =\{q_1\} \tag{8.3}$$

$$\begin{aligned}\delta =&\{\{(q_0, a) \to q_0\},\\ &\{(q_0, b) \to q_1\},\\ &\{(q_1, b) \to q_1\}\}\end{aligned} \tag{8.4}$$



This FSA defines a language over an alphabet of two symbols, *a* and *b*. The transition function δ is written as a set of tuples: the tuple $\{(q_0, a) \to q_0\}$ says that if you are in state q_0 and you see symbol a, you can consume it and stay in q_0. Because each pair of initial state and symbol has at most one resulting state, this FSA is **deterministic**: each string ω induces at most one path. Note that δ does not contain any information about what to do if you encounter the symbol a while in state q_1. In this case, you are stuck, and cannot accept the input string.

What strings does this FSA accept? We begin in q_0, but we have to get to q_1, since this is the only final state. We can accept any number of a symbols while in q_0, but we require a b symbol to transition to q_1. Once there, we can accept any number of b symbols, but if we see an a symbol, there is nothing we can do. So the regular expression corresponding to the language defined by M_1 is a^*bb^*. To see this, consider what M_1 would do if it were fed each of the following strings: *aaabb*; *aa*; *abbba*; *bb*.

Regular languages* Can every formal language be recognized by some finite state automata? No. Finite state automata can only recognize **regular languages**. The classic example of a non-regular language is $a^n b^n$; this language includes only those strings that contain n copies of symbol a, followed by n copies of symbol b. The **pumping lemma** demonstrates that this language cannot be accepted by any FSA. The proof is by contradiction. Suppose M is an FSA that accepts the language $a^n b^n$. By definition M must have a finite number of states; if we choose a string $a^m b^m$ such that m is bigger than the number of states in M, then the path through M must contain a cycle, and the transitions on this cycle must accept only the symbol a. But if there is a cycle, then we can repeat the cycle any number of times, "pumping up" the number of a symbols in the string. The automaton M must therefore also accept strings $a^{m'} b^m$, with $m' > m$. But these strings are not in the language $a^n b^n$, so we arrive at a contradiction. The proof will be covered in detail by any textbook on theory of computation (e.g., Sipser, 2012).

Determinism

- In a deterministic (D)FSA, the transition function is defined so that $\delta : Q \times \Sigma \rightarrow Q$. This means that every pair of initial state and symbol can transition to at most one resulting state.
- In a nondeterministic (N)FSA, $\delta : Q \times \Sigma \rightarrow 2^Q$. This means that a pair of initial state and symbol can transition to multiple resulting states. As a consequence, an NFSA may have multiple paths to accept a given string.
- We can determinize any NFSA using the powerset construction, but the number of states in the resulting DFSA may be exponential in the size of the original NFSA.
- Any **regular expression** can be converted into an NFSA, and thus into a DFSA.

The English Dictionary as an FSA We can build a simple "chain" FSA which accepts any single word. So, we can define the English dictionary with an FSA. However, we can make this FSA much more compact. (see slides)

- Begin by taking the **union** of all of the chain FSAs by defining **epsilon transitions** (transitions that do not consume an input symbol) from the start state to chain FSAs for each word (5303 states / 5302 arcs using a 850 word dictionary of "basic English").

- Eliminate the epsilon transitions by pushing the first letter to the front (4454 states / 4453 arcs)
- **Determinize** (2609 / 2608)
- **Minimize** (744 / 1535). The cost of minimizing an acyclic FSA is $O(E)$. This data structure is called a **trie**.

Operations In discussing talked about three operations: union, determinization and minimization. Other important operations are:

Intersection only accept strings in both FSAs: $\omega \in (M_1 \cap M_2)$ iff $\omega \in M_1 \cap \omega \in M_2$.

Negation only accept strings not accepted by FSA M: $\omega \in (\neg M)$ iff $\omega \notin M$.

concatenation accept strings of the form $\omega = [\omega_1 \omega_2]$, where $\omega_1 \in M_1$ and $\omega_2 \in M_2$.

FSAs are **closed** under all these operations, meaning that resulting automaton is still an FSA (and therefore still defines a regular language).

FSAs for Morphology

Now for some morphology. Suppose that we want to write a program that accepts only those words that are constructed in accordance with English derivational morphology:

- *grace, graceful, gracefully*
- *disgrace, disgraceful, disgracefully, ...*
- *Google,Googler,Googleology,...*
- **gracelyful, *disungracefully, ...*

As we saw in the English dictionary example, we could just make a list, and then take the union of the list using ϵ-transitions. The list would get very long, and it would not account for productivity (our ability to make new words like *antiwordificationist*). So let's try to use finite state machines instead. Our FSA will have to encode rules about morpheme ordering, called *morphotactics*.

Every word must have a stem, so we do not want to accept proposed words like *dis-* or *-ly*. This suggests that we should have at least two states: one for before we have seen a stem, and one for after. Assuming the alphabet Σ consists of all English morphemes, we can define a transition function so that it is only possible to transition from q_0 to q_1 by consuming a stem morpheme; by defining

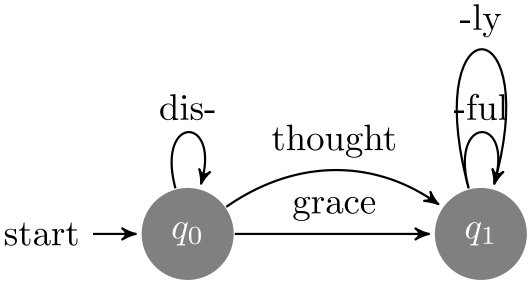


Figure 8.1: First try at modeling English morphology

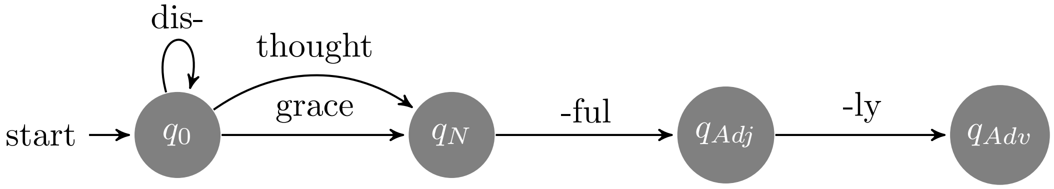


Figure 8.2: Second try at modeling English morphology, this time distinguishing parts-of-speech

$F = \{q_1\}$, we can ensure that every word has a stem. For prefixes, we can allow self-transitions in q_0 on prefix morphemes; we can do the same in q_1 for suffix morphemes.

The resulting FSA is shown in Figure 8.1 will accept *grace*, *disgrace*, *graceful disgraceful*, and even *disgracefully* (with two self-transitions in q_1). However, it will also accept **gracelyful* and **gracerly*. To deal with these cases, we need to think about what the suffixes are doing. The suffix *-ful* converts the noun *grace* into an adjective *graceful*; it does the same for words like *thoughtful* and *sinful*. The suffix *-ly* converts the adjective *graceful* to the adverb *gracefully* (to see the difference, compare *the ballet was graceful* to *the ballerina moved gracefully*.) These examples suggest that we need additional states in our FSA, such as q_{noun}, $q_{\text{adjective}}$, and q_{adverb}. Each of these is a potential final state, and the suffixes allow transitions between them. This FSA is shown in Figure 8.2.

However, with a little more thought, we see that this approach is still too simple. First, not every noun can be made into an adjective: **chairful* and **monkeyful* are perhaps suggestive of some kind of poetic meaning, but would not be recognized as standard English. Second, many nouns are made into adjectives using different suffixes, such as *music+al*, *fish+y*, and *elv+ish*. We need to create additional noun states to distinguish these noun groups, so as to avoid accepting

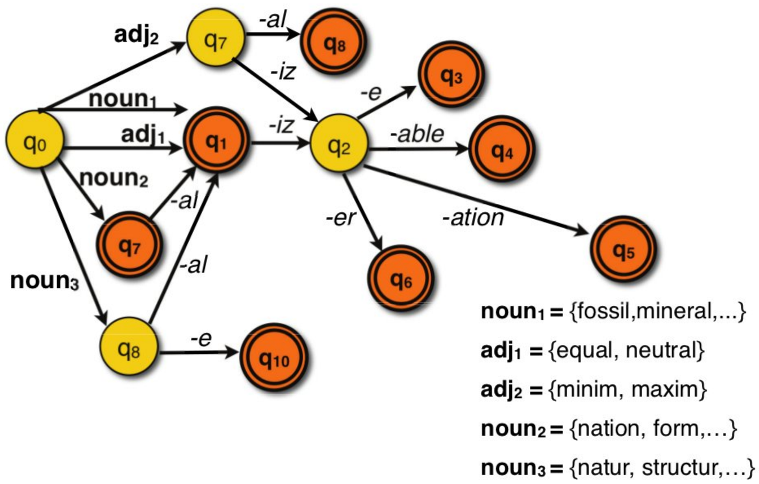


Figure 8.3: A fragment of a finite-state acceptor for derivational morphology. From Julia Hockenmaier's slides.

ill-formed words like **musicky* and **fishful*. We could continue to refine the FSA, coming ever-closer to an accurate model of English morphotactics. A fragment of such an FSA is shown in Figure 8.3.

This approach makes a key assumption: every word is either in or out of the language, with no wiggle room. Perhaps you agreed that *musicky* and *fishful* were not valid English words; but if forced to choose, you probably find *a fishful stew* or *a musicky tribute* preferable to *behaving disgracelyful*. To take the argument further, here are some Google counts for various derivational forms:

- *superfast*: 70M; *ultrafast*: 16M; *hyperfast*: 350K; *megafast*: 87K
- *suckitude*: 426K; *suckiness*: 378K
- *nonobvious*: 1.1M; *unobvious*: 826K; *disobvious*: 5K

Given this diversity of possible realizations of the same idea, rather than asking whether a word is **acceptable**, we might like to ask how acceptable it is. But finite state acceptors gives us no way to express *preferences* among technically valid choices. We will need to augment the formalism for this.

8.2 Weighted Finite State Automata

A weighted finite-state automaton $M = \langle Q, \Sigma, \pi, \xi, \delta \rangle$ consists of:

- A finite set of states $Q = \{q_0, q_1, \ldots, q_n\}$
- A finite alphabet Σ of input symbols
- Initial weight function, $\pi : Q \to \mathbb{R}$
- Final weight function $\xi : Q \to \mathbb{R}$
- A transition function $\delta : Q \times \Sigma \times Q \to \mathbb{R}$

We have departed from the FSA formalism in three ways:

- Every state can be a start state, with score π_q.
- Every state can be an end state, with score ξ_q.
- Transitions are possible between any pair of states on any input, with a score δ_{q_i,ω,q_j}.

Now, we can score every path through a weighted finite state acceptor (WFSA) by the sum of the weights for the transitions, plus the scores for the initial and final states. The **shortest path algorithm** finds the minimum-cost path through a WFSA for a string ω, with time complexity $\mathcal{O}(E + V \log V)$, where E is the number of edges and V is the number of vertices (Cormen et al., 2009).

Weighted finite state automata (WFSAs) are a generalization of unweighted FSAs: for any FSA M we can build an equivalent WFSA by setting $\pi_q = \infty$ for all $q \neq q_0$, $\xi_q = \infty$ for all $q \notin F$, and δ_{q_i,ω,q_j} for all transitions $\{(q_1, \omega) \to q_2\}$ that are not permitted by the transition function of M.

Applications of WFSAs

We can use WFSAs to score derivational morphology as suggested above. But let's start with some simpler examples.

Edit distance

An **edit distance** is a function of two strings, which quantifies their similarity: for example, *she* and *he* differ by only the addition of a single letter, while *you* and *me* differ on every letter. There are a huge number of ways to compute edit

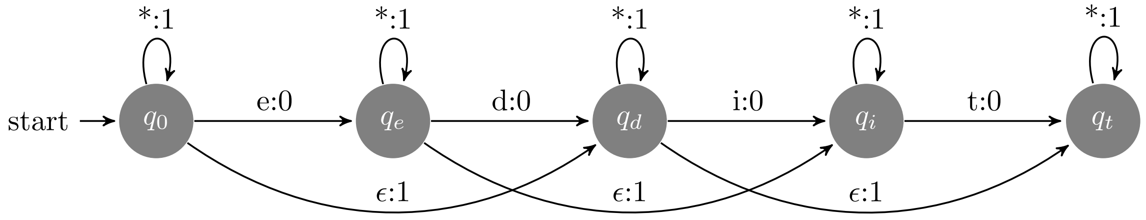


Figure 8.4: A weighted finite state acceptor for computing edit distance from the word *edit*.

distance (Manning et al., 2008), with applications in information retrieval, bioinformatics, and beyond.

Here we consider a simple edit distance, which computes the minimum number of character insertions, deletions, and substitutions required to get from one word to another. Insertions and deletions are penalized by a cost of one; substitutions have a cost of two. To compute this cost, we build a WFSA with one state for every letter in the word, plus an initial state q_0: for example, for the word *edit*, we build a machine with states q_0, q_e, q_d, q_i, q_t.

- The initial cost for q_0 is zero; for every other state, the initial cost is infinite.
- The final cost for q_t is zero; for every other state, the final cost is infinite.
- We define the transition function as follows:
  - The cost for "correct" symbols and rightward moves is zero: for example, $\delta_{q_0,e,q_e} = 0$, and $\delta_{q_i,t,q_t} = 0$.
  - The cost for self-transitions is one, regardless of the symbol: for example, $\delta_{q_d,*,q_d} = 1$. These self-transitions correspond to **insertions**.
  - The cost for epsilon transitions to the right is one: for example, $\delta_{q_e,\epsilon,q_d} = 1$. These transitions correspond to **deletions**.
  - The cost of all other transitions is ∞.

The machine is shown in Figure 8.4. The total edit distance for a string is the *sum* of costs across the best path through machine. Note that we did not define a cost for **substitutions** (e.g., from *him* to *ham*), because substitutions can be performed by a combination of insertion and deletion, for a total cost of two. However, some edit distances assign a cost of one to substitutions; can you see how to modify the WFSA to compute such an edit distance?

N-gram language models

Weighted finite state acceptors can also be used to compute probabilities of sequences — for example, the probability of a word sequence from an n-gram language model. To do this, we define the states and transitions so that each transition is equal to a condition probability, $\delta_{q_i,\omega_m,q_j} = \mathrm{p}(q_i, \omega_m \mid q_j)$, so that the product is equal to the joint probability of the state sequence and the string,

$$\mathrm{p}(\boldsymbol{q}_{1:M}, \boldsymbol{\omega}_{1:M}) = \prod_{m}^{M} \mathrm{p}(q_m, \omega_m \mid q_{m-1}). \tag{8.5}$$

For example, to construct a unigram language model over a vocabulary $\mathcal{V}$ of size V, we need just a single state. All transitions are self-transitions, with probability equal to the unigram word probability, $\delta_{q_0,w,q_0} = \mathrm{p}_1(w)$.

To construct a bigram language model, we need to model the conditional probability $\mathrm{p}(w_m \mid w_{m-1})$. To do this in a WFSA, we must create V different states: one for each context. Then we define the transition function as,

$$\delta_{q_i,w_m,q_j} = \begin{cases} \mathrm{p}(w_m \mid w_{m-1} = i), & j = m \\ 0, & \text{otherwise}. \end{cases} \tag{8.6}$$

Because each state represents a context, we require the transition function to ensure that we are in the right state after observing w_m: thus, we assign zero probability to all other transitions. The start function captures the probability $\mathrm{p}(w \mid \Diamond)$, and the final state function captures the probability $\mathrm{p}(\blacklozenge \mid w)$. Thus, the bigram probability of any string is computed by the product of transition scores,

$$\mathrm{p}_2(\boldsymbol{w}_{1:M}) = \mathrm{p}(w_1 \mid \Diamond) \times \left(\prod_{m=2}^{M} \mathrm{p}(w_m \mid w_{m-1}) \right) \times \mathrm{p}(\blacklozenge \mid w_M) \tag{8.7}$$

$$= \pi_{w_1} \times \left(\prod_{m=2}^{M} \delta_{q_{w_{m-1}},w_m,q_{w_m}} \right) \times \xi_{w_M}. \tag{8.8}$$

Can you see how to construct a trigram language model in the same way?

Interpolated n-gram language model

Knight and May (2009) show how to implement an interpolated bigram/unigram language model using a WFSA. Recall that an interpolated bigram language model computes probability,

$$\hat{\mathrm{p}}(w_m \mid w_{m-1}) = \lambda \mathrm{p}_1(w_m) + (1 - \lambda)\mathrm{p}_2(w_m \mid w_{m-1}), \tag{8.9}$$

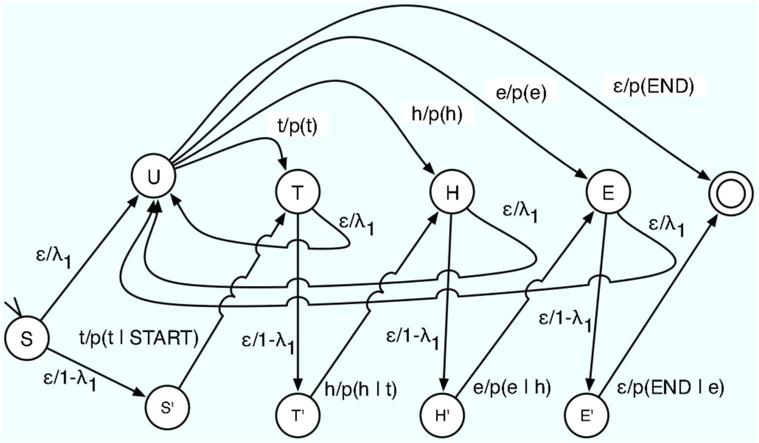


Figure 8.5: WFSA implementing an interpolated bigram/unigram language model (Knight and May, 2009). [todo: maybe redraw this for clarity?]

with p̂ indicating the interpolated probability, p_2 indicating the bigram probability, and p_1 indicating the unigram probability.

Note that Equation 8.9 involves both the multiplication and addition of probabilities. Knight and May (2009) achieve this through the use of **non-determinism**. The basic idea is shown in Figure 8.5. At each of the top row of states in Figure 8.5, there are two possible ϵ-transitions, which consume no input. With score λ, we transition to the generic state U, which "forgets" the local context; transitions out of U are scored according to the unigram probability model p_1. With score $1 - \lambda$, we transition to one of the context-remembering states, S', T', H', E'. Each of these states encodes the bigram context, and outgoing transitions are scored according to the bigram probability model p_2.

Any given path through this WFSA will have a score that multiplies together the probabilities of generating the words in the input, as well as the decisions about whether to use the unigram or bigram probability models. However, due to the non-determinism, each input string will have many possible paths to acceptance. Let's write these paths as sequences $z_1, z_2, \ldots, z_M$, with each $z_m \in \{1, 2\}$, indicating whether the unigram or bigram model was chosen to generating w_m.

Then the string *b,a* will have the following paths and scores:

$$\text{score}(1,1,1) = \lambda \times \text{p}_1(b) \times \lambda \times \text{p}_1(a) \times \lambda \times \text{p}_1(\blacklozenge) \tag{8.10}$$
$$= \lambda^3 \text{p}_1(a)\text{p}_1(b)\text{p}_1(\blacklozenge) \tag{8.11}$$
$$\text{score}(1,1,2) = \lambda^2(1-\lambda)\text{p}_1(b)\text{p}_1(a)\text{p}_2(\blacklozenge \mid a) \tag{8.12}$$
$$\text{score}(1,2,1) = \lambda^2(1-\lambda)\text{p}_1(b)\text{p}_2(a \mid b)\text{p}_1(\blacklozenge) \tag{8.13}$$
$$\text{score}(1,2,2) = \lambda(1-\lambda)^2\text{p}_1(b)\text{p}_2(a \mid b)\text{p}_2(\blacklozenge \mid a) \tag{8.14}$$
$$\text{score}(2,1,1) = \lambda^2(1-\lambda)\text{p}_2(b \mid \lozenge)\text{p}_1(a)\text{p}_1(\blacklozenge) \tag{8.15}$$
$$\text{score}(2,1,2) = \lambda^2(1-\lambda)\text{p}_2(b \mid \lozenge)\text{p}_1(a)\text{p}_2(\blacklozenge \mid a) \tag{8.16}$$
$$\text{score}(2,2,1) = \lambda^2(1-\lambda)\text{p}_2(b \mid \lozenge)\text{p}_2(a \mid b)\text{p}_1(\blacklozenge) \tag{8.17}$$
$$\text{score}(2,2,2) = (1-\lambda)^3\text{p}_2(b \mid \lozenge)\text{p}_2(a \mid b)\text{p}_2(\blacklozenge \mid a), \tag{8.18}$$

where $\lozenge$ is the special start symbol and $\blacklozenge$ is the special stop symbol. Each of these scores is a joint probability $\text{p}(\boldsymbol{w}_{1:M}, \boldsymbol{z}_{1:M})$; summing over them gives $\sum_{\boldsymbol{z}_{1:M}} \text{p}(\boldsymbol{w}_{1:M}, \boldsymbol{z}_{1:M}) = \text{p}(\boldsymbol{w}_{1:M})$, which is the desired marginal probability under the interpolated language model. Thus, in this case, we want not the score of the single best path, but the sum of the scores of **all** paths that accept a given input string.

8.3 Semirings

We have now seen three examples: an FSA for derivational morphology, and WFSAs for edit distance and language modeling. Several things are different across these examples.

Scoring

- In the derivational morphology FSA, we wanted a boolean "score": is the input a valid word or not?
- In the edit distance WFSA, we wanted a numerical (integer) score, with lower being better.
- In the interpolated language model, we wanted a numerical (real) score, with higher being better.

Nondeterminism

- In the derivational morphology FSA, we accept if there is any path to a terminating state.

- In the edit distance WFSA, we want the score of the single best path.
- In the interpolated language model, we want to sum over non-deterministic choices.

Semiring notation allows us to combine all of these different possibilities into a single formalism.

Formal definition

A semiring is a system $(\mathbb{K}, \oplus, \otimes, \overline{0}, \overline{1})$

- $\mathbb{K}$ is the set of possible values, e.g. $\{\mathbb{R}_+ \cup \infty\}$, the non-negative reals union with infinity
- $\oplus$ is an addition operator
- $\otimes$ is a multiplication operator
- $\overline{0}$ is the additive identity
- $\overline{1}$ is the multiplicative identity

A semiring must meet the following requirements:

- $(a \oplus b) \oplus c = a \oplus (b \oplus c)$, $(\overline{0} \oplus a) = a$, $a \oplus b = b \oplus a$
- $(a \otimes b) \otimes c = a \otimes (b \otimes c)$, $a \otimes \overline{1} = \overline{1} \otimes a = a$
- $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$, $(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$
- $a \otimes \overline{0} = 0 \otimes \overline{a} = \overline{0}$

Semirings of interest :

Name	$\mathbb{K}$	$\oplus$	$\otimes$	$\overline{0}$	$\overline{1}$	Applications
Boolean	$\{0, 1\}$	$\vee$	$\wedge$	0	1	identical to an unweighted FSA
Probability	$\mathbb{R}_+$	+	$\times$	0	1	sum of probabilities of all paths
Log-probability	$\mathbb{R} \cup -\infty \cup \infty$	$\oplus_{\text{log}}$	+	$-\infty$	0	log marginal probability
Tropical	$\mathbb{R} \cup -\infty \cup \infty$	min	+	∞	0	best single path

where $\oplus_{\text{log}}(a, b)$ is defined as $\log(e^a + e^b)$.

Semirings allow us to compute a more general notion of the "shortest path" for a WFSA.

- Our initial score is $\overline{1}$
- When we take a step, we use $\otimes$ to combine the score for the step with the running total.
- When nondeterminism lets us take multiple possible steps, we combine their scores using $\oplus$.

Example Let's see how this works out for our language model example.

$$\begin{aligned} score(\{a,b,a\}) = \overline{1} &\otimes \big(\lambda \otimes \mathrm{p}_2(a|*) \oplus (1-\lambda) \otimes \mathrm{p}_1(a)\big) \\ &\otimes \big(\lambda \otimes \mathrm{p}_2(b|a) \oplus (1-\lambda) \otimes \mathrm{p}_1(b)\big) \\ &\otimes \big(\lambda \otimes \mathrm{p}_2(a|b) \oplus (1-\lambda) \otimes \mathrm{p}_1(a)\big) \end{aligned}$$

Now if we plug in the **probability semiring**, we get

$$\begin{aligned} score(\{a,b,a\}) = 1 &\times \big(\lambda \mathrm{p}_2(a|*) + (1-\lambda)\mathrm{p}_1(a)\big) \\ &\times \big(\lambda \mathrm{p}_2(b|a) + (1-\lambda)\mathrm{p}_1(b)\big) \\ &\times \big(\lambda \mathrm{p}_2(a|b) + (1-\lambda)\mathrm{p}_1(a)\big) \end{aligned}$$

But if we plug in the **log probability semiring**, we need the edge weights to be equal to $\log \mathrm{p}_1$, $\log \mathrm{p}_2$, $\log \lambda$, and $\log(1-\lambda)$. Then we get:

$$\begin{aligned} score(\{a,b,a\}) = 0 &+ \log\big(\exp(\log\lambda + \log \mathrm{p}_2(a|*)) + \exp(\log(1-\lambda) + \log \mathrm{p}_1(a))\big) \\ &+ \log\big(\exp(\log\lambda + \log \mathrm{p}_2(b|a)) + \exp(\log(1-\lambda) + \log \mathrm{p}_1(b))\big) \\ &+ \log\big(\exp(\log\lambda + \log \mathrm{p}_2(a|b)) + \exp(\log(1-\lambda) + \log \mathrm{p}_1(a))\big) \\ = 0 &+ \log(\lambda \mathrm{p}_2(a|*) + (1-\lambda)\mathrm{p}_1(a)) \\ &+ \log(\lambda \mathrm{p}_2(b|a) + (1-\lambda)\mathrm{p}_1(b)) \\ &+ \log(\lambda \mathrm{p}_2(a|b) + (1-\lambda)\mathrm{p}_1(a)), \end{aligned}$$

which is exactly equal to the log of the score from the probability semiring.

- The score on any specific path will be the semiring **product** of all steps along the path.
- The score of any input will be the semiring **sum** of the scores of all paths that successfully process the input.
- What happens if we use the tropical semiring?

8.4 Finite state transducers

Finite state acceptors can determine whether a string is in a language, and weighted finite state acceptors can compute a score for every string from a given alphabet. We now consider a family of automata which can **transduce** one string into another. Formally, finite state transducers (FSTs) define **regular relations** over pairs of strings. We can think of them in two different ways:

- **Recognizer**: An FST accepts a pair of strings (input and output) if the pair is in the regular relation defined by the transducer.
- **Translator**: An FST takes an input string, and returns an output, such that the input/output pair is in the regular relation.

Like FSAs, finite-state transducers are defined as tuples. In this case, we define $M = \langle Q, \Sigma, \Delta, q_0, F, \delta, \sigma \rangle$, including:

- a finite set of states $Q = \{q_0, q_1, \ldots, q_n\}$;
- the finite alphabets Σ for input symbols and Δ for output symbols;
- an initial state $q_0 \in Q$, and a set of final states $F \subseteq Q$;
- a transition function $\delta : \langle Q \times \Sigma^* \rangle \rightarrow \langle Q \times \Delta^* \rangle$.

Example Consider the following FST, shown in Figure 8.6, which performs **pluralization** of some English words:

$$Q = \{q_0, q_{\text{regular}}, q_{\text{needs-e}}, q_{\text{pluralized}}\} \tag{8.19}$$

$$N = \{\textit{aardvark}, \ldots, \textit{wish}, \textit{wit}, \ldots, \textit{zyzzyva}^2\}(\text{the set of all English nouns}) \tag{8.20}$$

$$\Sigma = N \cup \{+\text{PL}\} \tag{8.21}$$

$$\Delta = N \cup \{+\textit{s}, +\textit{es}\} \tag{8.22}$$

$$q_0 = q_0 \tag{8.23}$$

$$F = \{q_{\text{regular}}, q_{\text{needs-e}}, q_{\text{pluralized}}\} \tag{8.24}$$

$$\begin{aligned}\delta = \{&(\langle q_0, \textit{aardvark}\rangle \rightarrow \langle q_{\text{regular}}, \textit{aardvark}\rangle),\\ &(\langle q_0, \textit{wish}\rangle \rightarrow \langle q_{\text{needs-e}}, \textit{wish}\rangle),\\ &(\langle q_0, \textit{wit}\rangle \rightarrow \langle q_{\text{regular}}, \textit{wit}\rangle),\\ &\ldots\\ &(\langle q_{\text{regular}}, +\text{PL}\rangle \rightarrow \langle q_{\text{pluralized}}, +\textit{s}\rangle)\\ &(\langle q_{\text{needs-e}}, +\text{PL}\rangle \rightarrow \langle q_{\text{pluralized}}, +\textit{es}\rangle)\end{aligned} \tag{8.25}$$

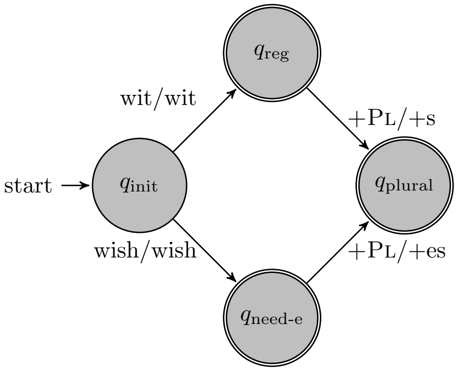


Figure 8.6: A finite state transducer for pluralizing English words.

This machine will accept the pairs $\langle$*wit*+PL, *wits*$\rangle$, $\langle$*wish*+PL, *wishes*$\rangle$, $\langle$*wit*, *wit*$\rangle$, but not the pairs $\langle$*wit*+PL, *wites*$\rangle$, $\langle$*wish*+PL, *wishs*$\rangle$, $\langle$*wish*+PL, *wish*$\rangle$. Thus, it correctly handles a small part of English orthography for pluralization; with a different word list, it could also be used to conjugate verbs to third-person singular. Consider how you might modify this FST to perform lemmatization.

Non-determinism Unlike non-deterministic finite state acceptors, not all non-deterministic finite state transducers (NFSTs) can be determinized. However, special subsets of NFSTs called **subsequential** transducers can be determinized efficiently (see 3.4.1 in Jurafsky and Martin (2009)).

8.5 Weighted FSTs

Weights can be added to FSTs in much the same way as they are added to FSAs. For any pair $\langle q \in Q, s \in \Sigma^* \rangle$, we have a set of possible transitions, $\langle q \in Q, t \in \Delta^*, \omega \in \mathbb{K} \rangle$, with a weight ω in the domain defined by the semiring. Table 8.1 shows the relationship between FSAs, FSTs, and their weighted generalizations.

	acceptor	transducer
unweighted	FSA: $\Sigma^* \to \{0, 1\}$	FST: $\Sigma^* \to \Sigma^*$
weighted	WFSA: $\Sigma^* \to \mathbb{K}$	WFST: $\Sigma^* \to \langle \Sigma^*, \mathbb{K} \rangle$

Table 8.1: A unified view of finite state automata

Example In section 8.2, we saw how to build an FSA that would compute the edit distance from any single word. With WFSTs, we can build a general edit distance computer, which computes the edit distance between any **pair** of words.

- $Q_0 \xrightarrow[a]{a} Q_0 : 0$
- $Q_0 \xrightarrow[\epsilon]{a} Q_0 : 1$
- $Q_0 \xrightarrow[a]{\epsilon} Q_0 : 1$

The shortest path for a pair of strings $\langle \boldsymbol{s}, \boldsymbol{t} \rangle$ in this transducer has a score equal to the minimum edit distance between the strings (in the tropical semiring). We can think of each path as defining a potential **alignment** between $\boldsymbol{s}$ and $\boldsymbol{t}$. That is, there are many ways to transduce *she* into *he*; in the minimum edit distance path, we have the alignment $\langle s, \epsilon \rangle, \langle h, h \rangle, \langle e, e \rangle$.

Operations on FSTs

FSTs are:

- Closed under **union**. If T_1 recognizes the relation R_1 and T_2 recognizes the relation R_2, then there exists an FST that recognizes the relation $R_1 \cup R_2$.
- Closed under **inversion**. If T_1 recognizes the relation $R_1 = \{s_i, t_i\}_i$, then there exists an FST that recognizes the relation defined by $\{t_i, s_i\}_i$, effectively switching the inputs and outputs.
- Closed under **projection**. If T_1 recognizes the relation $R_1 = \{s_i, t_i\}_i$, then there exist FSTs that recognize the relations defined by $\{s_i, \epsilon\}_i$ and $\{\epsilon, t_i\}_i$. Note that these relations ignore either the input or the output, and so are equivalent to finite state acceptors (FSAs).
- Not closed under **difference**, **complementation**, and **intersection**;
- Closed under **composition**, as described below.

FST composition is the basis for implementing the noisy channel model in FSTs, and can be used to support dozens of cool applications. Through composition, we can create finite state **cascades** that link together several simple models; closure guarantees that the resulting model is still a WFST.

Finite state composition

Suppose we have a transducer T_1 from Σ^* to Γ^*, and another transducer T_2 from Γ^* to Δ^*. Then the composition $T_1 \circ T_2$ is an FST from Σ^* to Γ^*. More formally,

Unweighted definition iff $\langle x, z \rangle \in T_1$ and $\langle z, y \rangle \in T_2$, then $\langle x, y \rangle \in T_1 \circ T_2$.

Weighted definition

$$(T_1 \circ T_2)(x, y) = \bigoplus_{z \in \Sigma^*} T_1(x, z) \otimes T_2(z, y) \tag{8.26}$$

> Note that weighted composition in the Boolean semiring is identical to unweighted composition.

Designing algorithms for automatic FST composition is relatively straightforward if there are no epsilon transitions; otherwise it's more challenging (Allauzen et al., 2009). Luckily, software toolkits like OpenFST take care of this for you.

Example

- $T_1 : Q_0 \xrightarrow[a]{x} Q_0, Q_0 \xrightarrow[b]{y} Q_0$
- $T_2 : Q_1 \xrightarrow{a} Q_1, Q_1 \xrightarrow{b} Q_2, Q_2 \xrightarrow{b} Q_2$
- $T_1 \circ T_2 : Q_1 \xrightarrow{x} Q_1, Q_1 \xrightarrow{y} Q_2, Q_2 \xrightarrow{y} Q_2$

For simplicity T_2 is written as a finite-state acceptor, not a transducer. Acceptors are a special case of transducers, where the output alphabet is $\Delta = \{\epsilon\}$.

8.6 Applications of finite state composition

Edit distance

Consider the general edit distance computer developed in section 8.5. It assigns scores to pairs of strings. If we compose it with an FSA for a given string (e.g., *tech*), we get a WFSA, who assigns score equal to the minimum edit distance from *tech* for the input string.

- Composing an FST with a FSA yields a FSA.

- A very useful design pattern is to build a **decoding** WFSA by composing a general-purpose WFST with an unweighted FSA representing the input.
- The best path through the resulting WFSA will be the minimum cost / maximum likelihood decoding.

Transliteration

English is written in a Roman script, but many languages are not. **Transliteration** is the problem of converting strings between scripts. It is especially important for names, which don't have agreed-upon translations.

A simple transliteration system can be implemented through the noisy-channel model.

- T_1 is an English character model, implemented as a transducer so that strings are scored as $\log \text{p}_r(c_1, c_2, \ldots, c_M)$.
- T_2 is a character-to-character transliteration model. This can be based on explicit rules,[3] or on conditional probabilities $\log \text{p}_t(c^{(f)} \mid c^{(r)})$.
- T_3 is an acceptor for a given string that is to be transliterated.

The machine $T_1 \circ T_2 \circ T_3$ scores English character strings based on their orthographic fluency (T_1) and adequacy (T_2).

Suppose you were given an Roman-script character model and a set of foreign-script strings, but no equivalent Roman-script strings. How would you use EM to learn a transliteration model?

Knight and May (2009) provide a more complex transliteration model, which transliterates between Roman and Katakana scripts, using a deep cascade that includes models of the underlying phonology. In their model,

Word-based translation

Machine translation can be implemented as a finite-state cascade. A simple approach is to compose three automata:

- T_1 is a language model, implemented as a transducer, where every path inputs and outputs the same string, with a score equal to $\log \text{p}(w_1, w_2, \ldots, w_M)$. This model's responsibility is to tell us that p(*Coffee black me pleases much*) $\ll$ p(*I like black coffee a lot*).

[3] http://en.wikipedia.org/wiki/Romanization_of_Russian

- T_2 is the translation machine. It contains a single state, and every transition takes a word from the source language and outputs a word in the target language. The weights are typically set to $\text{p}(w^{(t)} \mid w^{(s)})$. This model should assign a high probability to p(*cafe* | *coffee*), and a low probability to p(*cafe* | *tea*).

  Suppose we are translating Spanish to English. Then T_1 maps from English to English, since it is a language model in English; T_2 maps from English to Spanish. By the definition of finite state composition (Equation 8.26), the scores of the paths through these two transducers will be combined with the $\otimes$ operator; in the probability semiring, this means we will compute $\text{p}(\boldsymbol{w}^{(e)})\text{p}(\boldsymbol{w}^{(s)} \mid \boldsymbol{w}^{(e)}) = \text{p}(\boldsymbol{w}^{(s)}, \boldsymbol{w}^{(e)})$.

- T_3 is a deterministic finite-state acceptor, which accepts only the sentence to be translated. By composing $T_1 \circ T_2 \circ T_3$, we get a weighted finite-state acceptor for sentences in the target language (in our example, English).

  Recall that the composition $T_1 \circ T_2$ represents the joint probability $\text{p}(\boldsymbol{w}^{(s)}, \boldsymbol{w}^{(e)})$. The effect of T_3 is to "lock" $\boldsymbol{w}^{(s)}$ to the sentence to be translated. The shortest path in the composed machine $T_1 \circ T_2 \circ T_3$ thus computes,

$$\hat{\boldsymbol{w}}^{(e)} = \text{argmax}\, \boldsymbol{w}^{(e)} \text{p}(\boldsymbol{w}^{(s)}, \boldsymbol{w}^{(e)}) \tag{8.27}$$

$$= \text{argmax}\, \boldsymbol{w}^{(e)} \text{p}(\boldsymbol{w}^{(e)} \mid \boldsymbol{w}^{(s)}), \tag{8.28}$$

  which is the maximum-likelihood translation.

- Finally, note that we will need to allow ϵ-transitions in the translation model to handle cases like the translation of *mucho* to *a lot*. This introduces non-determinism to the finite-state cascade; again, we can think of this in terms of possible **alignments** between the source and target languages. The shortest-path algorithm computes the maximum likelihood translation while implicitly summing over all alignments.

8.7 Discriminative structure prediction

Now suppose we would like to use perceptron to learn to perform morphological segmentation. Imagine we are given a set of words $\boldsymbol{x}_{1:N}$ and their true segmentations $\boldsymbol{y}_{1:N}$. We would like to use perceptron to learn the weights of a WFST. How can we do it?

Recall that perceptron relies on computing a feature function $\boldsymbol{f}(\boldsymbol{x}, \boldsymbol{y})$. We will make this feature vector exactly equal to the finite-state transitions taken in the shortest-path transduction of $\boldsymbol{x}$ to $\boldsymbol{y}$. That is, each potential transition $(Q_i, \omega) \to Q_o$ corresponds to some entry j in the vector $\boldsymbol{f}(\boldsymbol{x}, \boldsymbol{y})$, and the value $f_j(\boldsymbol{x}, \boldsymbol{y})$ is equal to the number of times that transition was taken. Although FSTs can manipulate arbitrarily long strings, there will still be only a finite number of possible transitions, since both the state space and the alphabet are finite. The scores for these transitions can then be formed into the vector of weights $\boldsymbol{\theta}$, so that the score of the best path from $\boldsymbol{x}$ to $\boldsymbol{y}$ can be represented as the inner product $\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{y})$.

Let these transitions be represented in the weighted FST T. Given an instance $\boldsymbol{x}$, we build a chain acceptor $A_{\boldsymbol{x}}$. By composing T and $A_{\boldsymbol{x}}$, we obtain a WFSA in which the shortest path corresponds to the prediction $\hat{\boldsymbol{y}}$, and the transitions on this path are the feature vector $\boldsymbol{f}(\boldsymbol{x}, \hat{\boldsymbol{y}})$. We then compute the score of the best scoring path for accepting the true $\boldsymbol{y}$ segmentation in this machine; the transitions on this path form the feature vector $\boldsymbol{f}(\boldsymbol{x}, \boldsymbol{y})$. Given these two feature vectors, the perceptron update is as usual: $\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} + \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{y}) - \boldsymbol{f}(\boldsymbol{x}, \hat{\boldsymbol{y}})$. Weight averaging and passive-aggressive can be applied here, just as they were applicable in straightforward classification.

But unlike classification, we have now learned a function for making predictions over an **infinite set of labels**: all possible morphological segmentations for all possible words. We were able to do this by designing a feature function that shares features across different labels: if $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$ are nearly the same, then they will involve many of the same finite-state transitions, and so the feature vector $\boldsymbol{f}(\boldsymbol{x}, \boldsymbol{y})$ and $\boldsymbol{f}(\boldsymbol{x}, \hat{\boldsymbol{y}})$ will be nearly the same too. This is a powerful idea that will enable us to apply the tools of classification to a huge range of problems in language technology, including part-of-speech tagging, parsing, and even machine translation.

Chapter 9

Morphology

So far we have been focusing on NLP at the word level. Now we will explore meaning **inside of words**. We've already hinted at a morphological problem by introducing the idea of **lemmas**, where *serve/served/serving* all have the lemma *serve*.

From the perspective of document classification, these multiple forms may just seem like an annoyance, which we can get rid of by lemmatization or stemming (more on this later). But morphology conveys information which can be crucial for some applications.

Information retrieval With a search query like *bagel*, we want to get hits for the **inflected** form *bagels*; the same goes for irregular inflections like *corpus/corpora*, *goose/geese*. In **query expansion**, the search query is expanded to include all inflections of the search terms. Note that this isn't always what we want: for example, given a query for *Apple*, we may not want hits for *apples*.

Information extraction A major goal of information extraction is to capture references to events, and their properties. Event timing is conveyed in morphology: in English, we have suffixes for past tense (*she talk**ed***), the past participle (*she had spok**en***), and the present participle (*she is speaking*). Other languages can indicate many more details about event timing through morphology; for example, Romance languages like French have a much larger inventory of verb endings:

J'achete un velo	I buy a bicycle (now)
J'acheterai un velo	I will buy a bicycle
J'achetais un velo	I was buying a bicycle
J'ai acheté un velo	I bought a bicycle
J'acheterais un velo	I would buy a bicycle

In English, this function is mostly filled by auxiliary verbs like *will*, *was*, *had*, and *would*. This makes morphological analysis relatively less important for English, as we can get a long way with carefully constructed n-gram patterns (Riloff, 1996). But in languages like French and Spanish — where second-language learners are tormented by conjugation tables with dozens of different inflections — there seems little alternative to morphological analysis if language technology is to generalize across many verbs.

Document classification Even document classification tasks, such as sentiment analysis, are potentially impacted by morphology. For example, suppose you are doing sentiment analysis, and you encounter the out-of-vocabulary words *unfriended*, *antichrist*, *unputdownable*, or *disenchanted*. As unknown words, they would make no contribution to the overall sentiment polarity in a bag-of-words system. But with some morphological reasoning, we can see that they are indeed strongly subjective.

Translation In addition to recognizing morphology, there are applications in which we need to produce it. Translation is a classic case, especially when translating from morphologically simple languages like English and Chinese to morphologically rich languages, like French, Czech, German, and Swahili. Here again, a purely word-based approach would suffer from data sparsity: relatively rare words would be unlikely to be seen in every inflection, and thus the translation system would be unable to produce them.

Morphology, Orthography, and Phonology

Morphology interacts closely with two related systems: orthography and phonology. The **surface form** of a word is the form that is written down or spoken. This form results from the interactions between morphology and the orthographic and phonological systems. More specifically:

- **Morphology** describes how meaning is constructed from combining affixes. For example, it is a morphological fact of English that adding the affix +*S* to

Surface form	lemma	features
duck	*duck*	NOUN+SINGULAR
ducks	*duck*	NOUN+PLULAR
duck	*duck*	VERB+PRESENT
ducks	*duck*	VERB+THIRDPERSON+PRESENT

Table 9.1: Fragment of a morphologically-aware dictionary

many nouns creates a plural.

$$\textit{berry} + \textsc{Plural} \rightarrow \textit{berry+s}$$

Morphological rules may also include stem changes, such as *goose*+PLURAL → *geese*.

- **Orthography** specifically relates to writing. For example,

  $$\textit{berry+s} \rightarrow \textit{berries}$$

  is an orthographic rule. We have lots of these in English, which is one reason English spelling is difficult.

- **Phonology** describes how sounds combine. For example, the different pronunciations of the final *s* in *cats* (s) and *dogs* (z) follow from a phonological rule (Bender, 2013, example 25, page 30).

In English, morphologically distinct words may be pronounced differently even when they are spelled the same, and this can reflect morphological differences. *read*+PRESENT vs. *read*+PAST. Conversely, morphological variants may be spelled differently even when they sound the same, like *The Champions' league* versus *The Champion's league* versus *The Champions league*.

Productivity

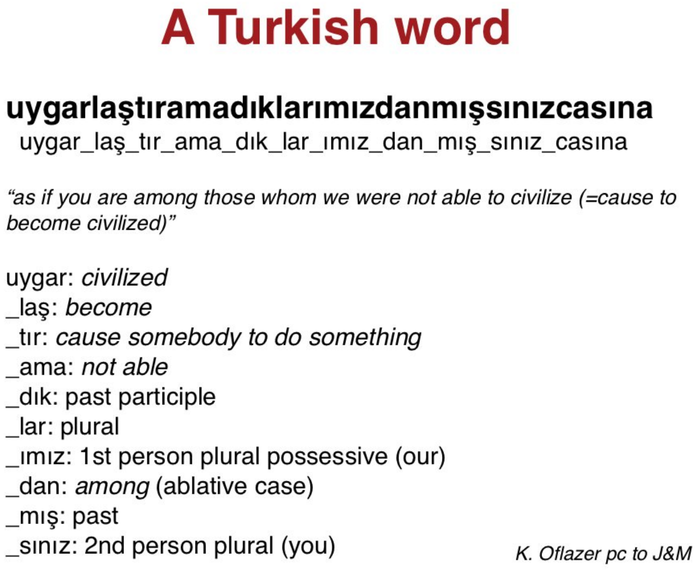
One idea for dealing with morphology is to build a morphologically-aware dictionary. The keys in this dictionary would correspond to **surface forms**, such as *served*. The values would include both the underlying **lemma** as well as any morphological features: in this case, the lemma is **serve**, and the feature is PAST. Given such a dictionary, we simply look up each surface form that we encounter.

As shown in the example in Table 9.1, we may need multiple entries for the same surface form; this means that there is ambiguity, so simple lookup will not suffice. Still another problem is that morphology is **productive**, meaning that it applies to new words. If you only know the words *Google* or *iPad*, you can immediately understand their inflected forms.

- Have you Googled that yet?
- I have broken all three iPads.

Derivational morphology (more on this later) is productive in another way: you can produce new words by applying morphological changes to existing words. hyper+un+desire+able+ity

In some languages, derivational morphology can create extremely complicated words. Jurafsky and Martin (2009) have a fun example from Turkish:



A Turkish word

uygarlaştıramadıklarımızdanmışsınızcasına
uygar_laş_tır_ama_dık_lar_ımız_dan_mış_sınız_casına

"as if you are among those whom we were not able to civilize (=cause to become civilized)"

uygar: *civilized*
_laş: *become*
_tır: *cause somebody to do something*
_ama: *not able*
_dık: past participle
_lar: plural
_ımız: 1st person plural possessive (our)
_dan: *among* (ablative case)
_mış: past
_sınız: 2nd person plural (you)

K. Oflazer pc to J&M

Figure 9.1: From (Jurafsky and Martin, 2009)

In the homework, you'll see examples from Swahili, which also has complex morphology. A dictionary of all possible surface forms in such languages would be gargantuan. So instead of building a static dictionary, we will try to model the underlying morphological and orthographic rules.

9.1 Types of morphemes

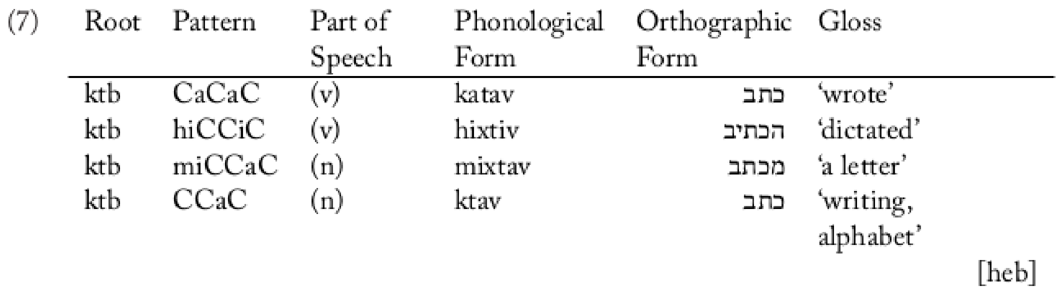
There are two broad classes of morphemes: **stems** and **affixes**. Intuitively, stems are the "main" part of meaning, and affixes are the modifiers. Typically, **stems** can appear on their own (they are **free**) and affixes cannot (they are **bound**).

Affixes can be categorized by where they appear with respect to the stem.

- **Prefixes**: *un+learn*, *pre+view*.
    - These examples are **derivational**, in that they form new words, rather than forming grammatical variants of the same word (*inflectional* morphology; more on this in section 9.2).
    - English has no inflectional prefixes, but other languages do. For example, in Swahili, *u-na-kata* means *you are cutting*, while *u-me-kata* means you have cut. In this example, *na* and *me* are prefixes, *kata* is the root.[1]
- **Suffixes** are the typical way of inflecting words in English, and in other languages in the Indo-European family. For example, in English: *I learn+ed*, *She learn+s*, *three apple+s*, *four fox+es*. English suffixes can also be derivational: for example: *modern+ity*, *fix+able*, and *deriv+ation+al*.
- **Circumfixes** go around the stem.
    - German has a circumfix for the past participle: *sagen* (say) → *ge+sag+t* (said)
    - English has a very small number of circumfix examples: *bold* → *em+bold+en*, and, arguably, *light* → *en+light+en*. Both of these examples are derivational.
    - French negation can be seen as a circumfix: *Je mange+NEG*→*Je ne mange pas* (I do not eat).[2]
    - More generally, morphemes can be non-contiguous, e.g. (Bender, 2013, example 7, page 12):

[1] Would it be better to think about *u*, *na*, and *me* as words? This example suggests that the word/affix distinction is not always clear-cut.

[2] In spoken French, the *ne* is gradually disappearing, so that *Je mange pas* is now acceptable.



(7)

Root	Pattern	Part of Speech	Phonological Form	Orthographic Form	Gloss
ktb	CaCaC	(v)	katav	כתב	'wrote'
ktb	hiCCiC	(v)	hixtiv	הכתיב	'dictated'
ktb	miCCaC	(n)	mixtav	מכתב	'a letter'
ktb	CCaC	(n)	ktav	כתב	'writing, alphabet'

[heb]

In this example, the root *ktb* (related to writing) is combined with patterns that indicate where to insert vowels to produce different parts-of-speech and meanings.

- **Infixes** go inside the stem.
  - In Tagalog (spoken in the Philippines), the root *hingi* indicates a request, and the infix *um* creates *humingi*, as in *I asked*.
  - English, *absolutely+fucking*→

    (9.1) *absofuckinglutely*

    (9.2) ?*absfuckingsolutely*

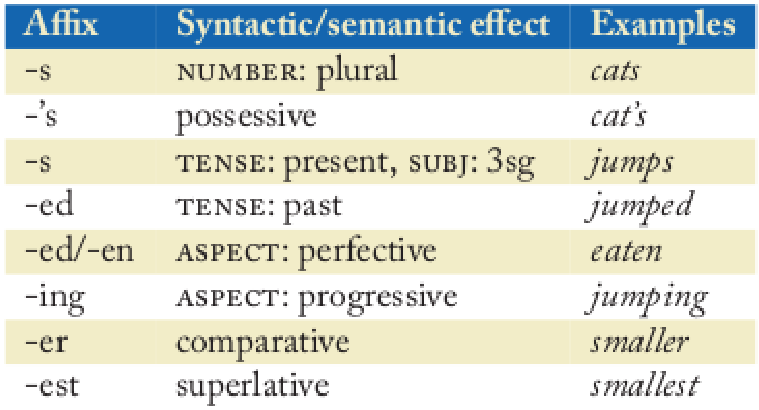
  where the '?' prefix indicates questionable linguistic acceptability.
- Morphology may be **non-segmental**, meaning that it doesn't involve any affix at all. For example, the pluralization of *goose* to *geese* is not accomplished through any affix, but through vowel alteration; the past tense marking of *eat* → *ate* is another example of this phenomenon, known as *apophony*. Languages in which morphemes are represented by affixes that are "glued together" (like *talk+ed* or *think+ing*) are known as **agglutinative**; languages in which morphemes are represented by changes to spelling and sound are known as **fusional**.
- What about words like *fish*, which have the same form in both singular and plural? We say that this word has a **zero** plural.

9.2 Types of morphology

Morphology serves a variety of linguistic functions, and acts in a variety of ways. Inflectional and derivational morphology are distinguished by their function; other forms of morphology, such as cliticization and compounding are distinguished by how they work. In this section, we will focus mainly on inflectional and derivational morphology, describing their roles in English, and in other languages when there is no adequate example in English.

Inflectional morphology

Inflectional morphology adds information about the stem, typically grammatical properties such as tense, number, and case. English has a relatively simple system of inflectional morphology, compared to many other languages.



Affix	Syntactic/semantic effect	Examples
-s	NUMBER: plural	*cats*
-'s	possessive	*cat's*
-s	TENSE: present, SUBJ: 3sg	*jumps*
-ed	TENSE: past	*jumped*
-ed/-en	ASPECT: perfective	*eaten*
-ing	ASPECT: progressive	*jumping*
-er	comparative	*smaller*
-est	superlative	*smallest*

Figure 9.2: From (Bender, 2013)

Nouns

English nouns are marked for **number** and **possession**. Number is typically marked by the suffix +*s*, e.g., *hat* + PLURAL → *hat*+*s*, but some words are pluralized differently, e.g., *geese*, *children*, and *fish*. Number is binary in English (singular versus plural), but many languages, such as Arabic and Sanskrit, include an additional **dual** number for groups of two. English has residual traces of the dual number, with *both* versus *all* and *either* versus *any*. Some Austronesian languages even have a **trial** number, for groups of three, and languages such as Arabic have a **paucal** number, for small groups. Conversely, nouns are not marked for number at all in Japanese and Indonesian.

Many languages mark nouns for **case**, which is the syntactic role that the noun plays in the sentence. In English, we do distinguish the case of some pronouns:

- *He (*NOMINATIVE*) gave her (*OBLIQUE*) his (*GENITIVE*) guitar.*
- *She gave him her guitar.*
- *I gave you our guitar.*
- *You gave me your guitar.*

The third person masculine pronoun appears as *he* in the nominative case, *him* in the oblique case, and *his* in the genitive case. English distinguishes these cases for all personal pronouns except for the second person, where the nominative and oblique cases are both *you*.

Other languages — such as Latin, Russian, Sanskrit, and Tamil — mark the case of all nouns. These languages have additional cases, such as dative (indirect object), accusative (direct object), and vocative (address). In German, noun is not inflected for case, but the articles and adjectives are, as shown in example 49 from Bender (2013):

(9.3) *Der alte Mann gab dem kleinen Affen die grosse Banane*.
The old man (NOM) gave the little monkey (DATIVE) the big banana (ACCUSATIVE)

Notice how *der*, *dem*, and *die* all mean the same thing (*the*), but they are spelled differently due to the case marking. The adjectives (*alte, kleinen, grosse*) are also marked for case.

Many languages — such as Romance languages — mark the gender and number of nouns by inflecting the article and adjective. e.g., Spanish:

(9.4) *El coche rojo pasó la luz roja*.
The red car ran the red light.

(9.5) *Los coches rojos pasó las luces rojas*.
The red cars ran the red lights.

Here, *la* is the feminine article and *el* is the masculine article; the adjective for *red* is inflected to *roja* when describing a feminine noun (*luz*, meaning light), and *rojo* when describing a masculine noun (*coche*, meaning car). The article and adjective must **agree** with noun for the sentence to be grammatical. The following examples are ungrammatical for this reason:

(9.6) **Los coches rojo pasó la luce rojas*

(9.7) **Los coches rojas pasó las luces rojos*

In English, demonstrative determiners mark number: e.g., *this book* vs *these books*, and the determiner and noun must agree, e.g. **this books*. Agreement is also required between subject and verb, as we will see shortly.

Romance languages like Spanish and French mark gender as masculine and feminine, but it need not be binary:

- English pronouns include neuter *it*; German, Sanskrit, and Latin do this for all nouns.
- Danish and Dutch distinguish **neuter** from **common** gender.[todo: example]
- Other languages distinguish **animate** and **inanimate** genders.

Verbs

English verbs are inflected for tense and number distinguishing past (*she acted*), present (*you act*), and third person singular (*she acts*). As with nouns, these inflections may change the orthography (*plan+ed* → *planned*), and there are many irregular patterns, e.g. *they eat / she eats / we ate*. English verbs are also inflected for aspect, distinguishing the perfective (*I had eaten*) and progressive (*I am eating*). The perfective and the past tense are identical for regular verbs, e.g. *we had talked*, *we talked*.

Many languages (e.g., Chinese and Indonesian), do not mark tense with morphology. For example, Indonesian uses function words rather than morphology to distinguish tense (Table 9.2).

Saya makan apel	I eat an apple
Saya sedang makan apel	I am eating an apple
Saya telah makan apel	I already ate an apple
Saya akan makan apel	I will eat an apple

Table 9.2: Indonesian uses function words (*sedang, telah, makan*) rather than morphology to distinguish verb tense. [todo: switch to exe]

Romance languages distinguish many more tenses than English with morphology. For example, Spanish has multiple past tenses: **preterite** and **imperfect**, distinguishing events that occurred at a specific past point in time from a continuous or repeated past state:

(9.8) *I ate onions yesterday*
Comí cebollas ayer

(9.9) *I ate onions every day*
Comía cebollas cada día

Spanish and French also have endings for conditional (*comería cebollas*, *I would eat onions*) and future (*comeré cebollas*, *I will eat onions*). In English, these differences are marked with time signals rather than morphology. In French and Spanish, time signals are also an option, e.g. *voy a comer cebollas*, which literally translates to *I am going to eat onions*.

Romance languages also have separate verb forms for every combination of number and person, while in English, only the third-person singular is distinguished:

- English: *I speak / you speak / she speak***s** */ we speak / you (pl) speak / they speak*
- Spanish: *Yo hablo / tu hablas / ella habla / nosotros hablamos / vosotros hableis / ellas hablan*
- French: *Je parle / tu parles / elle parle / nous parlons / vous parlez / ils parlent*

In Spanish and in many other Romance languages (but not French), the verb morphology is sufficiently descriptive that the subject is often omitted, since it can often be easily recovered from the verb ending and the context.

Other things can be marked with affixes, such as **evidentiality** – how the speaker came to know the information. In Eastern Pomo (a California language), there are verb suffixes for four evidential categories (McLendon, 2003):

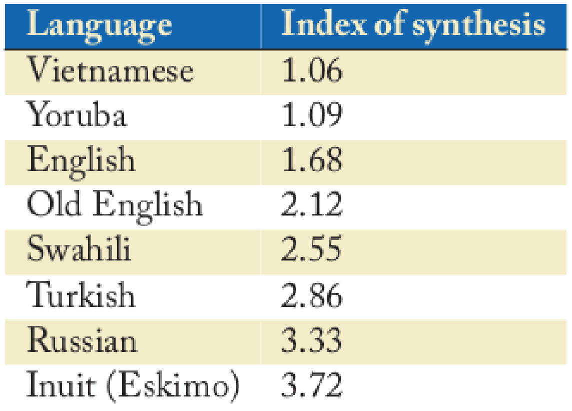
-ink'e	nonvisual sensory
-ine	inferential
-le	hearsay
-ya	direct knowledge

Adjectives and adverbs

Adjectives in English mark comparative and superlative (*taller*, *tallest*). Adverbs can mark comparative and superlative too: *Yangfeng paddles fast*, *Yi paddles faster*, *Uma paddles fastest*. As we have seen, adjectives can mark gender and number in languages like French and Spanish, where they are required to agree with the noun and determiner; adjectives also mark case in languages like German and Latin.

Synthetic and isolating languages

Languages with complex morphology are called **synthetic**; languages with simple morphology are called **isolating** or **analytic**. The **index of synthesis** quantifies this property by measuring the ratio of the number of morphemes in a given text to the number of words. On this index, English is relatively, but not extremely, analytic.



Language	Index of synthesis
Vietnamese	1.06
Yoruba	1.09
English	1.68
Old English	2.12
Swahili	2.55
Turkish	2.86
Russian	3.33
Inuit (Eskimo)	3.72

Figure 9.3: From Bender (2013)

An approximation of the index of synthesis is the type-token ratio. Can you see why? If you count the number of unique surface forms in 10K *parallel* sentences from a corpus of European Parliament transcripts, you get:

- English: 16k distinct word types
- French: 22k
- German: 32k
- Finnish: 55k

Derivational Morphology

Derivational morphology is a way to create new words and change part-of-speech.

- **nominalization**
  - *V + -ation: computerization*
  - *V + -er: walker*

  - *Adj + -ness: fussiness*
  - *Adj + -ity: obesity*

- **negation**: *undo, unseen, misnomer*
- **adjectivization**: *V + -able : doable, thinkable, N + -al : tonal, national, N + -ous: famous, glamorous*
- **abverbization**: *ADJ + -ily: clumsily*
- **lots more**: *rewrite, phallocentrism, ...*

You can create totally new words this way.
word → *wordify* → *wordification* → *wordificationism* → *antiwordificiationism* → *hyperantiwordificationism*

As with inflection, derivational morphology can require orthographic changes, e.g. *true+ly* → *truly* and *fussy+ness* → *fussiness*. It can also cause phonological changes, such as the change emphasis from *imPOSSible* to *impossiBILity*, and the change in vowel from *ferTILE* to *ferTILity*.

Other types of morphology

Cliticization combines *Georgia+'s* into *Georgia's*; the possessive clitic *'s* is syntactically independent but phonologically dependent. This syntactic independence can be seen in examples like (Bender, 2013, example 21):

(9.10) Jesse met the president of the university's cousin

In this example, the possessive modifies the *president*, but it attaches to the right edge of the entire noun phrase.

- Pronouns appear as clitics in French, e.g., *j'accuse* (I accuse), as does negation *Je n'accuse personne* (I don't accuse anyone).
- Another example is from Hebrew: *l'shana tova* (literally for year good, meaning happy new year); the preposition *for* appears as a clitic.

Compounding combines two words into a new word:

(9.11) *cream* → *ice cream*

We can think of *ice cream* as a word since it is a non-compositional combination of *ice* and *cream*. Perhaps someday the written space will be dropped, as it has been in *watermelon* (Figure 9.4).

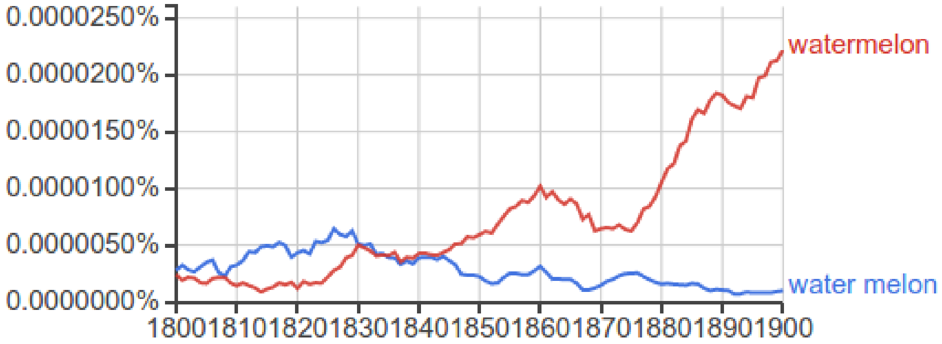


Figure 9.4: The written space in watermelon disappeared as the word became more frequent over the 19th century. From Google ngrams.

Portmanteaus combine words, truncating one or both.

(9.12) *smoke* + *fog* → *smog*

(9.13) *glass* + *asshole* → *glasshole*

Urban Dictionary is a fun source of contemporary portmanteaus.

Irregularities

English morphology contains a lot of irregularities: *know/knew/known*, *foot/feet*, *go/went*, etc. if you are not a native speaker, learning these was probably a pain in the neck. The good news is that there are fewer of these all the time! English is undergoing a process in which these irregular forms are gradually being replaced: for example, the past tense of *show* used to be *shew*, just as the past tense of *know* is still *knew* (Figure 9.5a). This transformation remains incomplete, as the past participle of *show* is still *shown*, and not *showed* (Figure 9.5b). However, this example points to the bad news for language learners: the most frequently-occuring words, like *know*, will be the last to change — if ever!

9.3 Computing and morphology

In this section, we will briefly overview some of the computational problems related to morphology. We don't yet have many tools to solve these problems, but we will soon: chapter 8 presents finite-state automata, which are the workhorse

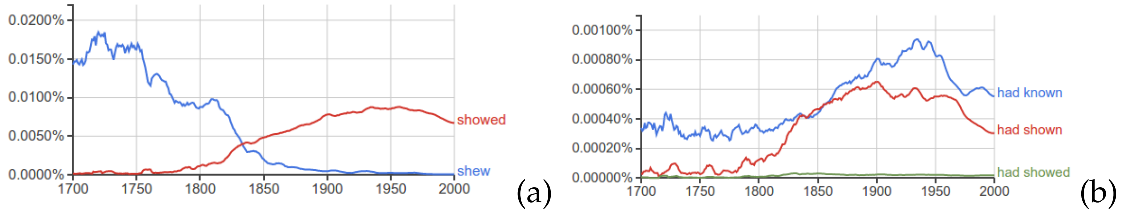


Figure 9.5: Google n-grams plots for inflections of *show*. While the past participle *had shown* is decreasing, this does not seem to be due to competition from the more regular *had showed*; rather, there appears to be a broader decrease in frequency of the past participle, shown by the parallel pattern for *had known*.

of morphological analysis in NLP. For now, we will simply state the problem definitions, and discuss some of the challenges involved.

Lemmatization

[todo: write]

Stemming

[todo: write]

Generation

[todo: write]

Normalization

[todo: write]

Chapter 10

Context-free grammars

So far we've explored finite-state models, which are capable of defining regular languages (and regular relations).

- **representations**: (weighted) finite state automata
- **probabilistic models**: HMMs (as a special case), CRFs
- **algorithms**: Viterbi, Forward-Backward, $\mathcal{O}(MK^2)$ time complexity.
- **linguistic phenomena**:
  - morphology
  - language models
  - part-of-speech disambiguation
  - named entity recognition (chunking)

Clearly there are formal languages that are not describable using finite-state machinery, such as the classic $a^n b^n$. But is the finite-state representation enough for natural language?

10.1 Is English a regular language?

In this section, we consider a proof that English is not regular, and therefore, no finite-state automaton could perfectly model English syntax. The proof begins by noting that regular languages are closed under **intersection**.

- $K \cap L$ is the set of strings in both K and L

- $K \cap L$ is regular iff K and L are regular

The proof strategy is as follows:

- Let K be the set of grammatical English sentences
- Let L be some regular language
- Show that the intersection is not regular

We're going to prove this using **center embedding**, as shown in the examples below:

(10.1) *The cat is fat.*

(10.2) *The cat that the dog chased is fat.*

(10.3) **The cat that the dog is fat.*

(10.4) *The cat that the dog that the monkey kissed chased is fat.*

(10.5) **The cat that the dog that the monkey chased is fat.*

Proof sketch:

- K is the set of grammatical english sentences.
  It excludes examples (10.3) and (10.5).
- L is the regular language *the cat* $(\textit{that } N)^{+} V_t^{+}$ *is fat*. It is crucial to see that this language is itself regular, and could be recognized with a finite-state acceptor.
- The language $L \cap K$ is *the cat* $(\textit{that } N)^{n} V_t^{n}$ *is fat*. This language is homomorphic to $a^n b^n$, which is known not to be regular. Since L is regular and $L \cap K$ is not regular, it follows that K cannot be regular.

It is important to understand that the issue is not just infinite repetition or productivity; FSAs can handle productive phenomena like *the big red smelly plastic figurine*. It is specifically the center-embedding phenomenon, because this leads to the same structure as the classic $a^n b^n$ language. What do you think of this argument?

Is deep center embedding really part of English?

Karlsson (2007) searched for multiple (phrasal) center embeddings in corpora from 7 languages:

- Very few examples of double embedding
- Only 13 examples of triple embedding (none in speech)
- Zero examples of quadruple embeddings

Note that we can build an FSA to accept center-embedding up to any finite depth. So in practice, we could build an FSA that accepts any center-embedded sentence that has ever been written. Does that defeat the proof? Chomsky and many linguists distinguish between

Competence the fundamental abilities of the (idealized) human language processing system;

Performance real utterances produced by speakers, subject to non-linguistic factors such as cognitive limitations.

Even if English *as performed* is regular, the underlying generative grammar may be context-free... **or beyond**.

How much expressiveness do we need?

Shieber (1985) makes a similar argument, showing that case agreement in Swiss-German cross-serial constructions is homomorphic to a formal language $wa^m b^n x c^m d^n y$, which is weakly non-context free. In response to the objection that all attested constructions are finite, Shieber writes:

> Down this path lies tyranny. Acceptance of this argument opens the way to proofs of natural languages as regular, nay, **finite**.

Regardless of what we think of these theoretical arguments, the fact is that in practice, many real constructions appear to be much simpler to handle in context-free rather than finite-state representations. For example,

(10.6) *The **processor has** 10 million times fewer transistors on it than todays typical microprocessors, **runs** much more slowly, and **operates** at five times the voltage...*

The verbs *has*, *runs*, and *operates* agree with the subject *the processor*; we want to accept this sentence, but reject all sentences in which this subject-verb agreement is lost. Handling this in a finite state representation would building separate components for third-person singular and non-third-person singular forms, and then replicating essentially all of verb-related syntax in each component. A **grammar** — formally defined in the next section — would vastly simplify things:

$$\begin{aligned}\text{S} &\rightarrow \text{NN VP}\\ \text{VP} &\rightarrow \text{VP3S} \mid \text{VPN3S} \mid \ldots\\ \text{VP3S} &\rightarrow \text{VP3S, VP3S, } \textit{and} \text{ VP3S} \mid \text{VBZ} \mid \text{VBZ NP} \mid \ldots\end{aligned}$$

10.2 Context-Free Languages

The Chomsky Hierarchy Every automaton defines a language, and different classes of automata define different classes of languages. The Chomsky hierarchy formalizes this set of relationships:

- **finite-state automata** define **regular** languages;
- **pushdown automata** define **context-free** languages;
- **Turing machines** define **recursively-enumerable** languages.

In the Chomsky hierarchy, context-free languages (CFLs) are a strict generalization of regular languages.

regular languages	**context-free languages**
regular expressions	context-free grammars (CFGs)
finite-state machines	pushdown automata
paths	derivations

Context-free grammars define CFLs. They are sets of permissible *productions* which allow you to **derive** strings composed of surface symbols. An important feature of CFGs is *recursion*, in which a nonterminal can be derived from itself.

More formally, a CFG is a tuple $\langle N, \Sigma, R, S \rangle$:

N a set of non-terminals
Σ a set of terminals (distinct from N)
R a set of productions, each of the form $A \rightarrow \beta$, where $A \in N$ and $\beta \in (\Sigma \cup N)^*$
S a designated start symbol

Context free grammars provide rules for generating strings.

- The left-hand side (LHS) of each production is a non-terminal $\in N$
- The right-hand side (RHS) of each production is a sequence of terminals or non-terminals, $\{n, \sigma\}^*, n \in N, \sigma \in \Sigma$.

A **derivation** t is a sequence of steps from S to a surface string $\boldsymbol{w} \in \Sigma^*$, which is the **yield** of the derivation. A derivation can be viewed as trees or as bracketings, as shown in Figure 11.4.

If there is some derivation t in grammar G such that $\boldsymbol{w}$ is the yield of t, then $\boldsymbol{w}$ is in the language defined by the grammar. Equivalently, for grammar G, we can write that $|\mathcal{T}_G(\boldsymbol{w})| \geq 1$. When there are multiple derivations of $\boldsymbol{w}$ in grammar G, this is a case of derivational **ambiguity**; if any such $\boldsymbol{w}$ exists, then we can say that the grammar itself is ambiguous.

Example The grammar below handles the case of center embedding:

$$\begin{aligned}
\text{S} &\rightarrow \text{NP VP}_1 && (10.1)\\
\text{NP} &\rightarrow \textit{the}\ \text{NP}\ \ |\ \text{NP}\ \textsc{RelClause} && (10.2)\\
\textsc{RelClause} &\rightarrow \textit{that}\ \text{NP V}_t && (10.3)\\
\text{V}_t &\rightarrow \textit{ate}\ |\ \textit{chased}\ |\ \textit{befriended}\ |\ \ldots && (10.4)\\
\text{N} &\rightarrow \textit{cat}\ |\ \textit{dog}\ |\ \textit{monkey}\ |\ \ldots && (10.5)\\
\text{VP}_1 &\rightarrow \textit{is fat} && (10.6)
\end{aligned}$$

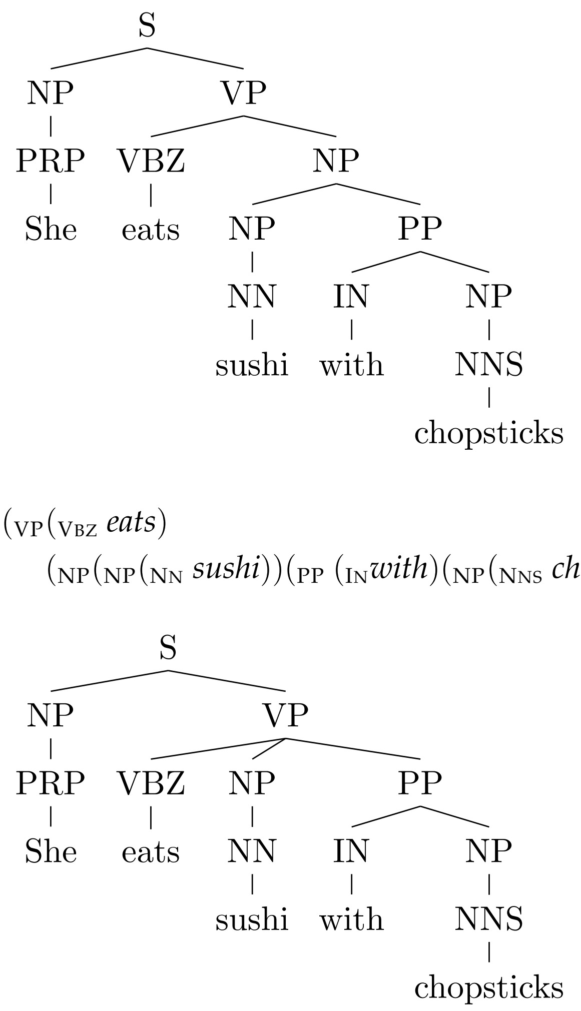
Here we are using a shorthand, where $\alpha \rightarrow \beta \mid \gamma$ implies two productions, $\alpha \rightarrow \beta$ and $\alpha \rightarrow \gamma$.

Semantics Ideally, each derivation will have a distinct semantic interpretation, and all possible interpretations will be represented in some derivation.

$$\begin{aligned}
&(_{\text{NP}}(_{\text{NP}}\ \textit{Ban}\ (_{\text{PP}}\ \textit{on}\ (_{\text{NP}}\ \textit{nude dancing}\ )))\\
&\quad(_{\text{PP}}\ \textit{on}\ (_{\text{NP}}\ \textit{Governor's desk}\ )))
\end{aligned}$$

$$\begin{aligned}
&(_{\text{NP}}\ \textit{Ban}\ (_{\text{PP}}\ \textit{on}\ (_{\text{NP}}(_{\text{NP}}\ \textit{nude dancing}\ )\\
&\qquad\qquad\qquad(_{\text{PP}}\ \textit{on}\ (_{\text{NP}}\ \textit{Governor's desk}\ )))))
\end{aligned}$$

In practice, this is quite hard to achieve with context-free grammars. For example, Johnson (1998) notes that there are three possible derivations for the verb phrase *ate dinner on the table with a fork*:



(S(NP(PRP *She*)(VP(VBZ *eats*)
(NP(NP(NN *sushi*))(PP (IN*with*)(NP(NNS *chopsticks*)))))))

(S(NP(PRP *She*)(VP(VBZ *eats*)
(NP(NN *sushi*))
(PP(IN*with*)(NP(NNS *chopsticks*))))))

Figure 10.1: Two derivations of the same sentence, shown as both parse trees and bracketings

"flat" *(ate dinner (on the table) (with a fork))*

"two-level" *((ate dinner) (on the table) (with a fork))*

"adjunction" *(((ate dinner) (on the table)) (with a fork))*

In this case, there doesn't seem to be any meaningful difference between these derivations. The grammar could avoid this problem by limiting its set of productions, but this change might cause problems in other cases.

10.3 Constituents

Our goal in using context-free grammars is usually not to determine whether a string is in the language defined by the grammar, but to acquire the derivation itself, which should explain the organization of the text and give some clue to its meaning. Therefore, a key question in grammar design is how to define the non-terminals.

Every non-terminal production **yields** a contiguous portion of the input string. For example, the VP non-terminal in Figure 11.4 (both parses) yields the substring *eats sushi with chopsticks*, and the PP non-terminal yields *with chopsticks*. These substrings, which are bracketed in the figure, are known as **constituents**. The main difference between the two parses in Figure 11.4 is that the second parse includes *sushi with chopsticks* as a constituent, and the first parse does not.

In a given string, which substrings should be constituents? Linguistics offers several tests for constituency, including: substitution, coordination, and movement.

Substitution

Constituents generated by the same non-terminal should be substitutable in many contexts:

(10.7) ($_{\text{NP}}$ *The ban*) *is on the desk.*

(10.8) ($_{\text{NP}}$ *The Governor's desk*) *is on the desk.*

(10.9) ($_{\text{NP}}$ *The ban on dancing on the desk*) *is on the desk.*

(10.10) *($_{\text{PP}}$ *On the desk*) *is on the desk.*

A more precise test for whether a set of substrings constitute a single category is whether they can be replaced by the same pronouns.

(10.11) ($_{\text{NP}}$ *It*) *is on the desk.*

What about verbs?

(10.12) *I* ($_{\text{V}}$ *gave*) *it to Anne.*

(10.13) *I* ($_{\text{V}}$ *taught*) *it to Anne.*

(10.14) *I* ($_{\text{V}}$ *gave*) *Anne a fish*

(10.15) **I* ($_{\text{V}}$ *taught*) *Anne a fish*

This suggests that *gave* and *taught* are not substitutable. We might therefore need non-terminals that distinguish verbs based on the arguments they can take. The technical name for this is *subcategorization*.

Coordination

Constituents generated by the same non-terminal can usually be *coordinated* using words like *and* and *or*:

(10.16) *We fought* ($_{\text{PP}}$ *on the hills*) *and* ($_{PP}$ *in the hedges*).

(10.17) *We fought* ($_{\text{ADVP}}$ *as well as we could*).

(10.18) **We fought* ($_{\text{ADVP}}$ *as well as we could*) *and* ($_{\text{PP}}$ *in the hedges*).

Like all such tests, coordination does not always work:

(10.19) *She* ($_{\text{VP}}$ *went*) ($_{\text{PP}}$ *to the store*).

(10.20) *She* ($_{\text{VP}}$ *came*) ($_{\text{PP}}$ *from the store*).

(10.21) *She* ($_{?}$ *went to*) *and* ($_{?}$ *came from*) *the store.*

Typically we would not think of *went to* and *came from* as constituents, but they can be coordinated.

Movement Valid constituents can be moved as a unit, preserving grammaticality. There are a number of ways in which such movement can occur in English.

Passivization (10.22) *(The governor) banned (nude dancing on his desk)*

(10.23) *(Nude dancing on his desk) was banned by (the governor)*

Wh- movement (10.24) *(Nude dancing was banned) on (the desk).*

(10.25) *(The desk) is where (nude dancing was banned)*

Topicalization (10.26) *(He banned nude dancing) to appeal to conservatives.*

(10.27) *To appeal to conservatives, (he banned nude dancing).*

10.4 A simple grammar of English

A goal of grammar design is to thread the line between two potential problems:

Overgeneration deriving strings that are not grammatical.

Undergeneration failing to derive strings that are grammatical.

To avoid undergeneration in a real language, we would need thousands of productions. Designing such a large grammar without overgeneration is extremely difficult.

Typically, grammars are defined in conjunction with large-scale **treebank** annotation projects.

- An annotation guideline specifies the non-terminals and how they go together.
- The annotators then apply these guidelines to data.
- The grammar rules can then be read off the data.

The Penn Treebank (PTB) contains one million parsed words of Wall Street Journal text (Marcus et al., 1993).

In the remainder of this section, we consider a small grammar of English.

Noun phrases

Let's start with noun phrases:

(10.28) ***She*** *sleeps* (Pronoun)

(10.29) ***Arlo*** *sleeps* (Proper noun)

These examples suggest that pronouns and proper nouns are substitutable, so we can define a production,

$$\text{NP} \rightarrow \text{P}\textsc{rp} \mid \text{N}\textsc{np}, \tag{10.7}$$

where NP stands for **noun phrase**. In this grammar, we will treat part-of-speech tags as the terminal vocabulary, but we could easily extend this to words by defining productions,

$$\text{P}\textsc{rp} \rightarrow \textit{she} \mid \textit{he} \mid \textit{I} \mid \textit{you} \ldots \tag{10.8}$$

$$\text{N}\textsc{np} \rightarrow \textit{Arlo} \mid \textit{Abigail} \ldots \tag{10.9}$$

What else could be a noun phrase?

(10.30) ***A lobster*** *sleeps*

(10.31) ***The lobster*** *sleeps*

(10.32) ***Lobsters*** *sleep*

(10.33) *****Lobster*** *sleeps*

The first two examples show that we can have common nouns (NN) as long as they are preceded by determiners (DT). We can also have plural nouns (NNS). But we cannot have common nouns **without** determiners — the final example doesn't work unless *Lobster* is a proper name.

We can handle these cases by defining a new nonterminal, NOM, which stands for **nominal**. A nominal is a constituent that cannot be a noun phrase by itself, but requires a determiner. We then add two productions:

$$\text{NP} \rightarrow \text{DT NOM} \mid \text{NNS} \tag{10.10}$$

$$\text{NOM} \rightarrow \text{NN} \mid \text{NNS} \tag{10.11}$$

Notice that these productions also allow *The lobsters sleep*, using the NOM → NNS production.

Noun phrases may also contain various **modifiers**.

(10.34) ***The blue fish*** *sleeps* (adjective)

(10.35) ***The four crabs*** *sleep* (cardinality)

We could try to handle these cases by adding to the nominal productions,

$$\text{NOM} \rightarrow \text{JJ NOM} \mid \text{CD NOM} \tag{10.12}$$

where JJ is an adjective and CD is a **cardinality**. Note that these productions are **recursive**, because NOM appears on the right-hand side. This means we can use the production to create a nominal with an infinite number of modifiers. This works for adjectives (*the angry blue plastic lobster*), but not for cardinals: **the four three crabs* is ungrammatical, so this grammar now **overgenerates**. We would need to further refine the grammar to handle this case properly, as well as to avoid **undergenerating** cases like *four crabs sleep*.

Modifiers can also come at the end of the noun phrase:

(10.36) *The girl from Omaha sleeps* (prepositional phrase)

(10.37) *Cats in Catalonia cry* (prepositional phrase)

(10.38) *The student who ate 15 donuts sleeps* (relative clause)

(10.39) *Mary from Omaha sleeps*

(10.40) *Cats who are in Catalonia cry*

(10.41) ?*Mary who ate 15 donuts sleeps*

These examples suggest that **prepositional phrases** (*from Omaha*, *in Catalonia*) can be attached to the end of any noun phrase. For **relative clauses** (*. . . who ate 15 donuts*), the situation is somewhat less clear. If we accept examples like (10.41), then we can handle both of these cases by adding the following NP productions,

$$\text{NP} \rightarrow \text{NP PP} \mid \text{NP } \textsc{RelClause} \tag{10.13}$$

We again have recursion: because the NP tag appears on the right side of the production, it is possible generate infinitely long noun phrases, like *the student from the city in the state below the river . . . .*

So overall, we can summarize the NP fragment of the grammar as,

$$\begin{aligned} \text{NP} \rightarrow & \text{PRP} \mid \text{NNP} \mid \text{DT } \textsc{Nom} \mid \text{NP PP} \mid \text{NP } \textsc{RelClause} \\ \textsc{Nom} \rightarrow & \textsc{Nn} \mid \textsc{AdjP Nom} \mid \textsc{Cd Nns} \mid \textsc{Nns} \end{aligned}$$

Are we done? Not close. We still haven't handled cardinal numbers in satisfactory way, and we are leaving out important details like number agreement, causing the grammar to overgenerate examples like *Mary sleep*. The process of grammar design would involve continuing to probe at the grammar with these sorts of examples until we handled as many as possible.

Adjectival and prepositional phrases

The noun phrase grammar mentioned prepositional phrases, such as

(10.42) *cats* ***from Catalonia***

(10.43) *pizza* ***in the refigerator***

(10.44) *pizza* ***in the old, broken refigerator***

(10.45) *the red switch* ***under the panel next to the radiator***

These examples suggest that prepositional phrases are formed by placing a preposition before any noun phrase — including noun phrases that already contain prepositional phrases, as in (10.45). This suggests the simple production,

$$\text{PP} \rightarrow \text{P NP}. \tag{10.14}$$

The noun phrase fragment also includes adjective modifiers, like *the blue lobster*. But in fact, adjectives can combine into phrases.

(10.46) *the **large blue** fish*

(10.47) *the **very funny** hat*

The first example, we have two adjectives; in the second, we have an adverb followed by an adjective. This suggests the following productions:

$$\text{ADJP} \rightarrow \textsc{Jj} \mid \textsc{Rb}\ \text{ADJP} \mid \textsc{Jj}\ \text{ADJP} \tag{10.15}$$

$$\text{NOM} \rightarrow \text{ADJP}\ \textsc{Nn} \mid \text{ADJP}\ \textsc{Nns} \tag{10.16}$$

Notice that if we instead added NOM → ADJP NOM, we would be introducing a considerable amount of ambiguity to the grammar. This would give us two different ways of generating multiple adjectives: by a series of NOM productions, or a series of ADJP productions. The proposed solution here increases the number of production rules, but decreases the number of ways to derive the same string.

Verb phrases

Let's now consider the verb and its modifiers.

(10.48) *She **sleeps***

(10.49) *She **sleeps restlessly***

(10.50) *She **sleeps at home***

(10.51) *She **eats sushi***

(10.52) *She **gives John sushi***

Each of these examples requires a production,

$$\text{VP} \rightarrow \text{V} \mid \text{VP}\ \textsc{Rb} \mid \text{VP PP} \mid \text{V NP} \mid \text{V NP NP} \tag{10.17}$$

But what about **She sleeps sushi* or **She speaks John Japanese*? We need a more fine-grained verb non-terminal to handle these cases.

$$\text{VP} \rightarrow \text{VP}\ \textsc{Rb} \mid \text{VP PP} \tag{10.18}$$

$$\text{VP} \rightarrow \text{V-}\textsc{intrans} \mid \text{V-}\textsc{trans}\ \text{NP} \mid \text{V-}\textsc{ditrans}\ \text{NP NP} \tag{10.19}$$

$$\text{V-}\textsc{intrans} \rightarrow \textit{sleeps} \mid \textit{talks} \mid \textit{eats} \mid \ldots \tag{10.20}$$

$$\text{V-}\textsc{trans} \rightarrow \textit{eats} \mid \textit{knows} \mid \textit{gives} \mid \ldots \tag{10.21}$$

$$\text{V-}\textsc{ditrans} \rightarrow \textit{gives} \mid \textit{tells} \mid \ldots \tag{10.22}$$

Notice that many verbs can be produced by multiple non-terminals: because we could have *Mary eats* and *Mary eats sushi*, we have to be able to derive *eats* from both V-INTRANS and V-TRANS.

To complete this fragment, we would also need to handle modal and auxiliary verbs that create complex tenses, like *She will have eaten sushi* but not **She will have eats sushi*.

Sentences

We can now define the part of the grammar that deals with entire sentences. Perhaps the simplest type of sentence includes a subject and a predicate,

(10.53) *She eats sushi*

To handle this we simply need,

$$\text{S} \rightarrow \text{NP VP}. \tag{10.23}$$

This rule can handle a number of other examples, like *she gives Alice the sushi*, *she eats*, etc. But things get more complex when we consider that sentences can be embedded inside other sentences:

(10.54) *Sometimes, she eats sushi*

(10.55) *In Japan, she eats sushi*

We therefore add two more productions,

$$\text{S} \rightarrow \text{ADVP S} \tag{10.24}$$

$$\text{S} \rightarrow \text{PP S} \tag{10.25}$$

What about **I eats sushi*, **She eat sushi*? To handle these, we need additional productions that enforce subject-verb agreement:

$$\text{S} \rightarrow \text{NP.3S VP.3S} \mid \text{NP.N3S VP.N3S}$$

In some languages, there are many other forms of agreement. **Feature grammars** provide a notation that can capture this kind of agreement, while remaining in the context-free class of languages.

Coordination

As mentioned above, one test for constituency is whether constituents of the same proposed type can be **coordinated** using words like *and* and *or*. For example,

(10.56) *She eats (sushi) and (candy)*

(10.57) *She (eats sushi) and (drinks soda)*

(10.58) *(She eats sushi) and (he drinks soda)*

(10.59) *(fresh) and (tasty) sushi*

These examples motivate, respectively, the following productions,

$$\begin{aligned}
\text{NP} &\rightarrow \text{NP } \textsc{Cc} \text{ NP} && (10.26)\\
\text{VP} &\rightarrow \text{VP } \textsc{Cc} \text{ VP} && (10.27)\\
\text{S} &\rightarrow \text{S } \textsc{Cc} \text{ S} && (10.28)\\
\text{ADJP} &\rightarrow \text{ADJP } \textsc{Cc} \text{ ADJP} && (10.29)\\
\textsc{Cc} &\rightarrow \textit{and} \mid \textit{or} \mid \ldots && (10.30)
\end{aligned}$$

We would need a little more cleverness to properly cover coordinations of more than two elements.

Odds and ends

Consider the example,

(10.60) *I gave sushi to the girl **who eats sushi**.*

This is a relative clause, which we already hinted at in the section on noun phrases. It requires its own non-terminal.

$$\begin{aligned}
\textsc{RelClause} &\rightarrow \textsc{Wp} \text{ VP} && (10.31)\\
\textsc{Wp} &\rightarrow \textit{who} \mid \textit{that} \mid \textit{which} \mid \ldots && (10.32)
\end{aligned}$$

Here are some related examples:

(10.61) *I took sushi from the man **offering sushi**.*

(10.62) *I gave sushi to the woman **working at home**.*

This is a gerundive postmodifier, which again requires its own non-terminal.

$$\begin{aligned}\text{NOM} \rightarrow & \text{NOM } \textsc{GerundVP} && (10.33)\\ \textsc{GerundVP} \rightarrow & \textsc{Vbg} \mid \textsc{Vbg} \text{ NP} \mid \textsc{Vbg} \text{ PP} \mid \ldots && (10.34)\\ \textsc{Vbg} \rightarrow & \textit{offering} \mid \textit{working} \mid \textit{talking} \mid \ldots && (10.35)\end{aligned}$$

Finally, we need to deal with questions, such as ***can*** *she eat sushi?* (and notice it's not *can she* ***eats*** *sushi*).

$$\begin{aligned}\text{S} \rightarrow & \textsc{Aux} \text{ NP VP} && (10.36)\\ \textsc{Aux} \rightarrow & \textit{can} \mid \textit{did} \mid \ldots && (10.37)\end{aligned}$$

Clearly this is just a small fragment of all the productions and non-terminals we would need to generate all observed English sentences. And as we will see, even this grammar fragment suffers from significant ambiguity. It is this issue that we will tackle in chapter 11.

10.5 Grammar equivalence and normal form

There may be many grammars that express the same context-free language.

- Grammars are **weakly equivalent** if they generate the same strings.
- Grammars are **strongly equivalent** if they generate the same strings **and** assign the same phrase structure to each string.

In Chomsky Normal Form (CNF), all productions are either:

$$\begin{aligned}A \rightarrow & BC\\ A \rightarrow & a\end{aligned}$$

All CFGs can be converted into a CNF grammar that is weakly equivalent — meaning that it generates exactly the same set of strings. As we will soon see, this conversion is very useful for parsing algorithms.

In CNF, all productions have either two or zero non-terminals on the right-hand size. To deal with productions that have more than two non-terminals on the RHS, we create new "dummy" non-terminals. For example, if we have $W \rightarrow X\ Y\ Z$, we can replace this production with two productions: $X \rightarrow W\ X\backslash W$ and $X\backslash W \rightarrow Y\ Z$, where $X\backslash W$ is a new dummy non-terminal. Figure 10.2 conveys this idea in a real example.

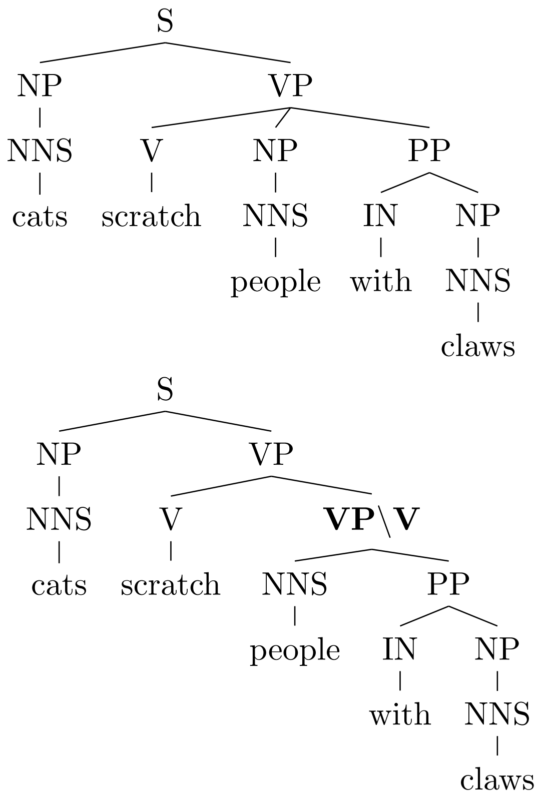


Figure 10.2: Binarization of the VP → V NP PP production

Note that *people with claws* was not a constituent in the original grammar, but it is a constituent in the binarized grammar. Therefore, after parsing it is important to take care to "un-binarize" the resulting parse.

What about unary productions, such as NP → NNS? While we could easily deal with this in the grammar, as we will see, in practice it is best dealt with by modifying the parsing algorithm itself.

Chapter 11

CFG Parsing

Parsing is the task of identifying the correct derivation for a sentence in a context-free language. Here are some possible approaches:

Top-down Start with the start symbol, and see if it is possible derive the sentence.

Bottom-up Combine the observed symbols using productions from the grammar, replacing them with the appropriate left-hand side. Continue applying this process until only the start symbol is left.

Left-to-right Move through the input, incrementally building a parse tree.

Before we get into these different possibilities, let us consider whether exhaustive search is possible. Suppose we only have one non-terminal, X, and it has binary productions

$$\begin{aligned} &\text{X} \rightarrow \text{X X} \\ &\text{X} \rightarrow \textit{the girl} \mid \textit{ate sushi} \mid \ldots \end{aligned}$$

How many different ways could we derive a sentence in this language? This is equal to the number of binary bracketings of the words in the sentence, which is a Catalan number. Catalan numbers grow **super-exponentially** in the length of the sentence, $C_n = \frac{(2n)!}{(n+1)!n!}$. As with sequence labeling, we cannot search the space of possible derivations naïvely; we will again rely on dynamic programming to search efficiently by reusing shared substructures.

11.1 CKY parsing

The CKY algorithm[1] is a bottom-up approach to parsing in a context free grammar. It efficiently tests whether a string is in a language, without considering all possible parses. The algorithm first forms small constituents, and then tries to merge them into larger constituents.

Let's start with an example grammar:

$$\begin{aligned}
\text{S} &\rightarrow \text{VP NP} \\
\text{NP} &\rightarrow \text{NP PP} \mid \textit{we} \mid \textit{sushi} \mid \textit{chopsticks} \\
\text{PP} &\rightarrow \text{P NP} \\
\text{P} &\rightarrow \textit{with} \\
\text{VP} &\rightarrow \text{V NP} \mid \text{V PP} \\
\text{V} &\rightarrow \textit{eat}
\end{aligned}$$

Suppose we encounter the sentence *We eat sushi with chopsticks*.

- The first thing that we notice is that we can apply unary terminal productions to obtain the part-of-speech sequence NP VP NP P NP.
- Next, we can apply a binary production to merge the first NP VP into an S.
- Or we could merge VP NP into VP ...
- ...and so on.

The CKY algorithm systematizes this approach, incrementally constructing a table t in which each cell $t[i, j]$ contains the set of nonterminals that can derive the span $w_{i:j-1}$. If $S \in t[0, M]$, then $\boldsymbol{w}$ is in the language defined by the grammar.

Algorithm 6 gives the details. We begin by filling in the diagonal: the entries $t[m, m+1]$ for all $m \in \{0 \ldots M-1\}$. These are filled with terminal productions that yield the individual tokens; for the word w_2 = *sushi*, we fill in $t[2, 3] = \{\text{N}\}$, and so on. Next we fill in cells spanning length 2: $t[0, 2], t[1, 3], \ldots, t[M-2, M]$. These are filled in by looking for binary productions capable of producing at least one entry in the cells corresponding to left and right children. Next we fill in cells spanning length 3, and so on. For each of these cells we have to search over the **split point** k, which divides the left and right children of the non-terminal that yields the entire span. Finally we arrive at $\ell = M$, which corresponds to the cell

[1] The name is for Cocke-Kasami-Younger, the inventors of the algorithm. It is sometimes called **chart parsing**, because of its chart-like data structure.

Algorithm 6 The CKY algorithm for parsing with context-free grammars

```
for $m \in \{0 \dots M-1\}$ do
    $t[m, m+1] \leftarrow \{X : X \rightarrow w_m \in R\}$
for $\ell \in \{2 \dots M\}$ do
    for $m \in \{0 \dots M-\ell\}$ do
        for $k \in \{m+1 \dots m+\ell-1\}$ do
            $t[m, m+\ell] \leftarrow t[m, m+\ell] \cup \{X : (X \rightarrow Y\ Z) \in R \wedge Y \in t[m,k] \wedge Z \in t[k, m+\ell]\}$
```

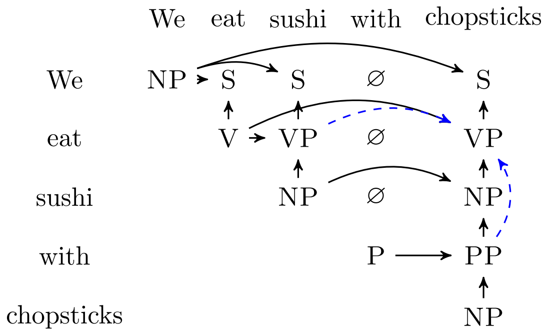


Figure 11.1: An example completed CKY chart. There are two paths to VPin position $t[1,5]$, one in black and another in dashed blue.

$t[0, M]$. If we can find a split point k such that we can produce an element in $t[0,k]$ and an element in $t[k, M]$ as productions from S, then we can successfully parse the sentence. Figure 11.1 shows the chart that arises from parsing the sentence *we eat sushi with chopsticks* using the grammar defined above.

The CKY algorithm assumes that all productions with non-terminals on the RHS are binary. What do we do when this is not true?

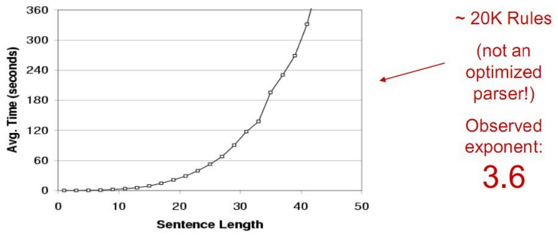
- For productions with more than two elements on the right-hand side, we binarize, creating additional non-terminals. For example, if we have the production VP → V NP NP (for ditransitive verbs), we might convert to VP → VP$_{ditrans}$/NP NP, and then add the production VP$_{ditrans}$/NP → V NP.
- What about unary productions like S → VP → V → *eat*? To handle this case, we compute the *unary closure* of each non-terminal. For example, if the grammar includes S → VP and VP → V, then we add S → V to the unary closure of S. Then for each entry $t[i,j]$ in the table, for each non-terminal $A \in t[i,j]$, we add all elements of the reflexive unary closure for A to $t[i,j]$.

Complexity

Space The space complexity is $\mathcal{O}(M^2\#|N|)$. We are building a table of size M^2, and each cell must hold up to $\#|N|$ elements, where $\#|N|$ is the number of non-terminals.

Time The time complexity is $\mathcal{O}(M^3\#|R|)$. At each cell, we search over $\mathcal{O}(M)$ split points, and $\#|R|$ productions, where $\#|R|$ is the number of production rules in the grammar.

Notice that these are considerably worse than the finite-state algorithms of Viterbi and forward-backward, which are linear time; generic shortest-path for finite-state automata has complexity $\mathcal{O}(M \log M)$. As usual, these are worst-case asymptotic complexities. But in practice, things can be worse than worst-case! (See Figure 11.2) This is because longer sentences tend to "unlock" more of the grammar — they involve non-terminals that do not appear in shorter sentences.



~ 20K Rules

(not an optimized parser!)

Observed exponent:
3.6

Figure 11.2: Figure from Dan Klein's lecture slides

11.2 Ambiguity in parsing

In many applications, we don't just want to know whether a sentence is grammatical, we want to know what structure is the best analysis. Unfortunately, syntactic ambiguity is endemic to natural language:[2]

Attachment ambiguity *we eat sushi with chopsticks*, *I shot an elephant in my pajamas.*

Modifier scope *southern food store*

Particle versus preposition *The puppy tore up the staircase.*

[2] Examples borrowed from Dan Klein's slides

Complement structure *The tourists objected to the guide that they couldn't hear.*

Coordination scope *"I see," said the blind man, as he picked up the hammer and saw.*

Multiple gap constructions *The chicken is ready to eat*

These forms of ambiguity can combine, so that a seemingly simple sentence like *Fed raises interest rates* can have dozens of possible analyses, even in a minimal grammar. Real-size broad coverage grammars permit millions of parses of typical sentences. Faced with this ambiguity, classical parsers faced a tradeoff:

- achieve broad coverage but admit a huge amount of ambiguity;
- or settle for limited coverage in exchange for constraints on ambiguity.

The problem of syntactic parsing is to find the best choice among the many legal parses for a given sentence. We will now explore some data-driven solutions to this problem.

Local solutions

Some ambiguity can be resolved locally. Consider the following examples,

(11.1) [*imposed* [*a ban* [*on asbestos*]]]

(11.2) [*imposed* [*a ban*][*on asbestos*]]

This is a case of attachment ambiguity: do we attach the prepositional phrase *on asbestos* to the verb *imposed*, or the noun phrase *a ban*. To solve this problem, Hindle and Rooth (1990) proposed a likelihood ratio test:

$$LR(v, n, p) = \frac{\mathrm{p}(p \mid v)}{\mathrm{p}(p \mid n)} = \frac{\mathrm{p}(\textit{on} \mid \textit{imposed})}{\mathrm{p}(\textit{on} \mid \textit{ban})} \tag{11.1}$$

where they select VERB attachment if $LR(v, n, p) > 1$.

But the likelihood-ratio approach ignores important information, like the phrase being attached.

(11.3) ...[*it* [*would end* [*its venture* [*with Maserati*]]]]

(11.4) ...[*it* [*would end* [*its venture*][*with Maserati*]]]

The likelihood ratio gets this example wrong,

- $\mathrm{p}(\textit{with} \mid \textit{end}) = \frac{607}{5156} = 0.118$

- $\text{p}(\textit{with} \mid \textit{venture}) = \frac{155}{1442} = 0.107$

Other features (e.g., *Maserati*) argue for noun attachment, since entities such as *Maserati* tend to participate in ventures, rather than being used as instruments to bring about an ending (which is what the verb phrase attachment implies). To combine these sorts of features into a single predictive model, we will need machine learning.

Machine learning solutions Ratnaparkhi et al. (1994) propose a classification-based approach, using logistic regression (maximum entropy):

$$\text{p}(\text{Noun attachment} \mid \textit{would end its venture with Maserati}) = \frac{e^{\boldsymbol{\theta}^\top \boldsymbol{f}(\text{noun-attach}, \textit{would end its venture with Maserati})}}{e^{\boldsymbol{\theta}^\top \boldsymbol{f}(\text{noun-attach}, \textit{would end its venture with Maserati})} + e^{\boldsymbol{\theta}^\top \boldsymbol{f}(\text{verb-attach}, \textit{would end its venture with Maserati})}}$$

Features include n-grams and word classes from hierarchical word clustering (see chapter 15); accuracy is roughly 80%.

Collins and Brooks (1995) argued that attachment depends on four **heads**:

- the preposition (*with*)
- the VP attachment site (*end*)
- the NP attachment site (*venture*)
- the NP to be attached (*Maserati*)

They propose a backoff-based approach:

- First, look for counts of the tuple $\langle$*with, Maserati, end, venture*$\rangle$
- If none, try $\langle$*with, Maserati, end*$\rangle$ + $\langle$*with,end,venture*$\rangle$ + $\langle$*with,Maserati, venture*$\rangle$
- If none, try $\langle$*with, Maserati*$\rangle$ + $\langle$*with,end*$\rangle$ + $\langle$*with,venture*$\rangle$
- If none, try $\langle$*with*$\rangle$

Accuracy of this method is roughly 84%. This approach of combining relative frequency estimation, smoothing, and backoff was very characteristic of 1990s statistical NLP.

Beyond local solutions

Framing the problem as attachment ambiguity is limiting. It assumes the parse is mostly done, leaving just a few attachment ambiguities to solve. But realistic sentences have more than a few syntactic interpretations, and attachment decisions are interdependent. For example, consider the sentence,

(11.5) *Cats scratch people with claws with knives.*

We may want to attach *with claws* to *scratch*, as would be correct in the sentence in *Cats scratch people with claws*. But then we have nowhere to attach *with knives*. Only by considering these decisions jointly can we make the right choice. The task of statistical parsing is to produce a single analysis that resolves all syntactic ambiguities.

11.3 Probabilistic Context-Free Grammars

In a **probabilistic context-free grammar** (PCFG), each production $X \to \alpha$ is associated with a probability $\text{p}(\alpha \mid X)$. These probabilities are conditioned on the left-hand side, so they must normalize to one over possible right-hand sides, $\sum_{\alpha'} \text{p}(\alpha' \mid X) = 1$. For example, for the verb phrase productions, we might have,

VP →V	0.3
VP →V NP	0.6
VP →V NP NP	0.1

which would indicate that transitive verbs are twice as common as intransitive verbs, which in turn are three times more common than ditransitive verbs.

Given probabilities on the productions, we can then score the probability of a derivation as a product of the probabilities of all of the productions. Consider the PCFG in Table 11.1 and the parse in Figure 11.3.

S	→ NP VP	0.9
S	→ S CC S	0.1
NP	→ N	0.2
NP	→ DT N	0.3
NP	→ N NP	0.2
NP	→ JJ NP	0.2
NP	→ NP PP	0.1
VP	→ V	0.4
VP	→ V NP	0.3
VP	→ V NP NP	0.1
VP	→ VP PP	0.2
PP	→ P NP	1.0

Table 11.1: A fragment of an example probabilistic context-free grammar (PCFG)

The probability of this parse is:

$$\begin{aligned}
\mathrm{p}(\tau, \boldsymbol{w}) =& P(\text{S} \rightarrow \text{NP VP}) \\
& \times P(\text{NP} \rightarrow \text{N}) \times P(\text{N} \rightarrow \textit{they}) \\
& \times P(\text{VP} \rightarrow \text{VP PP}) \\
& \times P(\text{VP} \rightarrow \text{V NP}) \times P(\text{V} \rightarrow \textit{eat}) \\
& \times P(\text{NP} \rightarrow \text{N}) \times P(\text{N} \rightarrow \textit{sushi}) \\
& \times P(\text{PP} \rightarrow \text{P NP}) \times P(\text{P} \rightarrow \textit{with}) \\
& \times P(\text{NP} \rightarrow \text{N}) \times P(\text{N} \rightarrow \textit{chopsticks}) && (11.2) \\
=& 0.9 \times 0.2 \times 0.2 \times 0.3 \times 0.2 \times 1.0 \times 0.2 \\
& \times \text{probability of terminal productions} && (11.3)
\end{aligned}$$

Now if we consider the alternative parse in which the prepositional phrase attaches to the noun, all of these probabilities are the same, with one exception: instead of the production VP → VP PP, we would have the production NP → NP PP. Since $P(\text{VP} \rightarrow \text{VP PP}) > P(\text{NP} \rightarrow \text{NP PP})$ in the PCFG, the verb phrase attachment would be preferred.

This example hints at a big problem with PCFG parsing on non-terminals such as NP, VP, and PP: we will **always** prefer either VP or PP attachment, without regard to what is being attached! This problem is addressed later in the chapter.

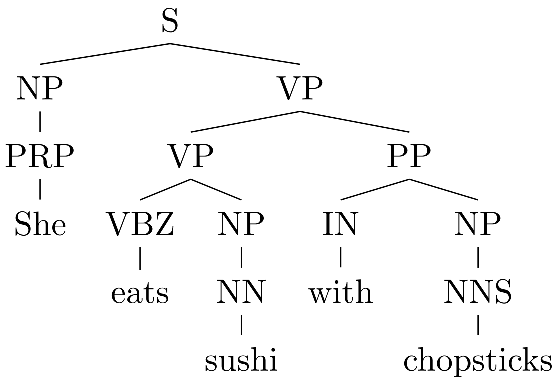


Figure 11.3: An example derivation

More formally, for a given sequence $\boldsymbol{w}$, we want to select the parse τ that maximizes $\text{p}(\tau \mid \boldsymbol{w})$.

$$\begin{aligned}\text{argmax}\,\tau \text{p}(\tau \mid \boldsymbol{w}) =& \text{argmax}\,\tau \frac{\text{p}(\tau, \boldsymbol{w})}{\text{p}(\boldsymbol{w})}\\ =& \text{argmax}\,\tau \text{p}(\tau, \boldsymbol{w})\\ =& \text{argmax}\,\tau \text{p}(\boldsymbol{w} \mid \tau)\text{p}(\tau)\\ =& \text{argmax}\,\tau : \boldsymbol{w} = \text{yield}(\tau)\text{p}(\tau)\end{aligned}$$

As in CFGs, the **yield** of a tree is the string of terminal symbols that can be read off the leaf nodes. The set $\{\tau : \boldsymbol{w} = \text{yield}(\tau)\}$ is exactly the set of all derivations of $\boldsymbol{w}$ in a CFG G.

Estimation

As in supervised HMMs, estimation is easy (for now!). We can estimate the production probabilites directly from a treebank, using relative frequency estimation. For example,

$$P(\text{VP} \rightarrow \text{VP PP}) = \frac{\text{count}(\text{VP} \rightarrow \text{VP PP})}{\text{count}(\text{VP})}$$

Three basic problems for PCFGs

Let $\tau \in T$ be a derivation, $\boldsymbol{w}$ be a sentence, and λ a PCFG.

- **Decoding**: Find $\hat{\tau} = \text{argmax}\,\tau \text{p}(\tau, \boldsymbol{w}; \lambda)$

	Sequences	Trees
model	HMM	PCFG
decoding	Viterbi algorithm	CKY
decoding complexity	$\mathcal{O}(M^2\lvert K\rvert)$	$\mathcal{O}(M^3\lvert R\rvert)$
likelihood	forward algorithm	inside algorithm
marginals	forward-backward	inside-outside

Table 11.2: Relationships between generative probabilistic models of sequences and trees

Algorithm 7 CKY algorithm with weighted productions

```
for $m \in \{0, \ldots, M-1\}$ do
    for all $X \in \text{tags}(w_j)$ do
        $t[m, m+1, X] \leftarrow P(X \rightarrow w_m)$
for $\ell \in \{2 \ldots M\}$ do
    for $m \in \{0, \ldots, M-\ell\}$ do
        for $k \in \{m+1, \ldots, m+\ell-1\}$ do
            for all $(X \rightarrow Y\ Z) \in R$ do
                $t[m, m+\ell, X] \leftarrow t[m, m+\ell, X] \bigoplus (\psi_{X \rightarrow Y\ Z} \otimes t[m, k, Y] \otimes t[k, m+\ell, Z])$
```

- **Likelihood**: Find $\text{p}(\boldsymbol{w}; \lambda) = \sum_{\tau} \text{p}(\tau, \boldsymbol{w}; \lambda)$
- **(Unsupervised) Estimation**: Find $\text{argmax}\, \lambda \text{p}(\boldsymbol{w}_{1 \ldots N} \mid \lambda)$

These three problems are analogous to the problems identified by Rabiner (1989) for Hidden Markov Models. More analogies between these models are identified in Table 11.2.

CKY with weights

It is not difficult to extend CKY to include probabilities or other weights. Let us write $\psi_{X \rightarrow Y\ Z}$ for the score for the production $X \rightarrow Y\ Z$. In the PCFG, this score is simply a probability, $\psi_{X \rightarrow Y\ Z} = P(X \rightarrow Y\ Z)$; in a more general **weighted context-free grammar** (WCFG), the score may be some other quantity, such as a log-potential score $\boldsymbol{\theta}^{\top} \boldsymbol{f}(X \rightarrow Y\ Z)$. Algorithm 7 shows how to perform CKY parsing in a WCFG.

In the boolean semiring, we have $\oplus = \vee$, $\otimes = \wedge$, and $\psi_{X \rightarrow Y\ Z} = \text{True}$ if $X \rightarrow Y\ Z$ is a production in the grammar. The $\oplus$ operation ensures that we take a disjunction over all split-points k and all children Y and Z; the $\otimes$ operations require that we can derive the span $\boldsymbol{w}_{m:k-1}$ from Y, **and** the span $\boldsymbol{w}_{k:m+\ell-1}$ from Z. Let's write $X \rightsquigarrow \boldsymbol{w}_{i:j}$ if it is possible to derive the substring $\boldsymbol{w}_{i:j}$ from the non-terminal X. If $Y \rightsquigarrow \boldsymbol{w}_{m:k-1}$ and $Z \rightsquigarrow \boldsymbol{w}_{k:m+\ell-1}$, and $X \rightarrow Y\ Z$ is in the grammar, then $X \rightsquigarrow \boldsymbol{w}_{m:m+\ell-1}$.

In the "tropical" probability semiring, we have $\oplus = \max$, $\otimes = \times$, and $\psi_{X \rightarrow Y\ Z} = P(X \rightarrow Y\ Z)$. Let's write $\psi(X \rightsquigarrow \boldsymbol{w}_{i:j})$ for the probability of the highest-probability derivation of $\boldsymbol{w}_{i:j}$ from the non-terminal X. Then,

$$\begin{aligned}
\text{if } t[Y, m, k] =& \psi(Y \rightsquigarrow \boldsymbol{w}_{m:k-1}) && (11.4)\\
\text{and } t[Z, k, m+\ell] =& \psi(Z \rightsquigarrow \boldsymbol{w}_{k:m+\ell-1}) && (11.5)\\
\text{then } \psi(X \rightsquigarrow \boldsymbol{w}_{m:m+\ell-1}) =& \max_{Y,Z,k} P(X \rightarrow Y\ Z) \times t[Y, m, k] \times t[Z, k, m+\ell] && (11.6)
\end{aligned}$$

The **inside algorithm** computes the probability of producing a span of text $\boldsymbol{w}_{i:j}$ from a non-terminal X. To do this, we move to a semiring where $\oplus = +$,

$$\begin{aligned}
t[X, i, j] =& \sum_{Y,Z,k} P(X \rightarrow Y\ Z) P(Y \rightarrow \boldsymbol{w}_{i:k}) P(Z \rightarrow \boldsymbol{w}_{k+1:j}) && (11.7)\\
=& P(X \rightarrow \boldsymbol{w}_{i:j}). && (11.8)
\end{aligned}$$

The relationship between CKY and the Inside Algorithm is perfectly analogous to the relationship between Viterbi and the Forward Algorithm, and is carried out by exactly the same change of semirings.

11.4 Parser evaluation

Before continuing to more advanced parsing algorithms, we need to consider how to measure parsing performance. Suppose we have a set of **reference parses** — the ground truth — and a set of **system parses** that we would like to score. A simple solution would be **per-sentence accuracy**: the parser is scored by the proportion of sentences on which the system and reference parses exactly match.[3] But we would like to assign *partial credit* for correctly matching parts of the reference parse. The PARSEval metrics do that, scoring each system parse via:

[3] Most parsing papers do not report results on this metric, but Finkel et al. (2008) find that a near-state-of-the-art parser finds the exact correct parse on 35% of sentences of length ≤ 40, and on 62% of parses of length ≤ 15 in the Penn Treebank.

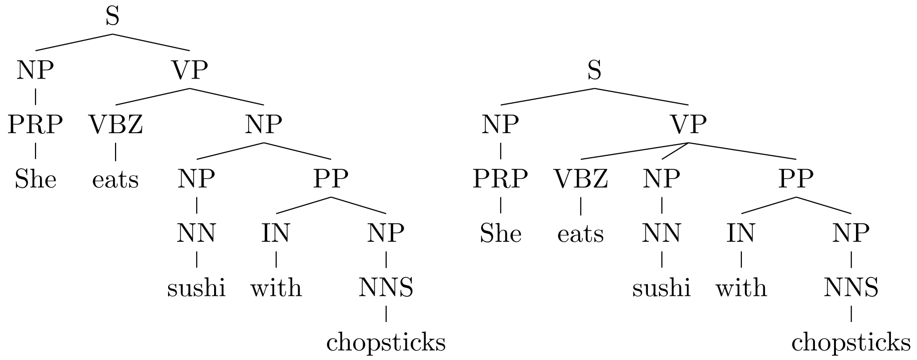


Figure 11.4: Suppose that the left parse is the system output, and the right parse is the ground truth; the precision is 0.75 and the recall is 1.0.

Precision, the fraction of brackets in the system parse that match a bracket in the reference parse.

Recall, the fraction of brackets in the reference parse that match a bracket in the system parse.

As in chapter 3, the F-measure is the harmonic mean of precision and recall, $F = \frac{2*P*R}{R+P}$.

In **labeled** precision and recall, the system must also match the non-terminals for each bracket; in **unlabeled** precision and recall, it is only required to match the bracketing structure.

In Figure 11.4, suppose the top tree is the system parse and the bottom tree is the reference parse. We have the following spans:

- S $\to \boldsymbol{w}_{1:5}$: true positive
- VP $\to \boldsymbol{w}_{2:5}$: true positive
- NP $\to \boldsymbol{w}_{3:5}$: false positive
- PP $\to \boldsymbol{w}_{4:5}$: true positive

So for this parse, we have a (labeled and unlabeled) precision of $\frac{3}{4} = 0.75$, and a recall of $\frac{3}{3} = 1.0$, for an F-measure of 0.86. The best automatic CFG parsers get an F-score of approximately 0.92 on the Penn Treebank (PTB) today (McClosky et al., 2006).

11.5 Improving PCFG parsing

Regardless of the parsing algorithm, pure PCFG parsing on Penn Treebank non-terminals (e.g., NP, VP) doesn't work well: Johnson (1998) shows that a PCFG estimated from treebank production counts gets an F-measure of only $F = 0.72$. Why?

Problems with PCFG parsing

Substitutability Recall that substitutability is a criterion for constituency. Are NPs really substitutable? No, because some pronouns cannot be both subjects and objects (Figure 11.5).

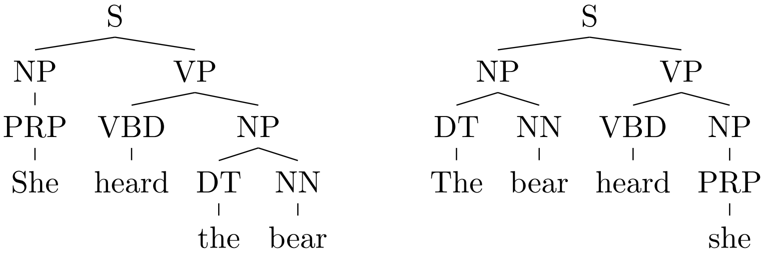


Figure 11.5: A grammar that allows *she* to take the object position wastes probability mass on ungrammatical sentences.

We might address this problem by **splitting** the NP tag into nominitive (*she*) and oblique (*her*) cases, but this distinction is only relevant for pronouns: other nouns can appear in either position.

A related point is that we have no flexibility on PP attachment. If $P(\text{NP} \rightarrow \text{NP PP}) > P(\text{VP} \rightarrow \text{VP PP})$, we will always prefer NP attachment; if not, we will always prefer VP attachment. More fine-grained NP and VP categories might allow us to make attachment decisions more accurately.

Semantic preferences In addition to grammatical constraints such as case marking, we have semantic preferences: for example, that conjoined entities should be similar. In Figure 11.6, you probably prefer the left parse, which conjoins *France* and *Italy*, rather than the right parse, which conjoins *wine* and *Italy*. But it is impossible for a PCFG to distinguish these parses! They contain exactly the same productions, so the resulting probabilities will be the same, no matter how you define the probabilities of each production.

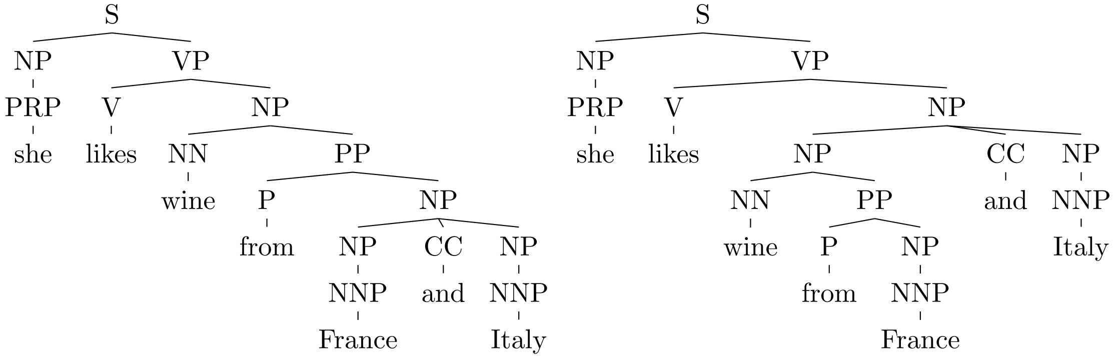
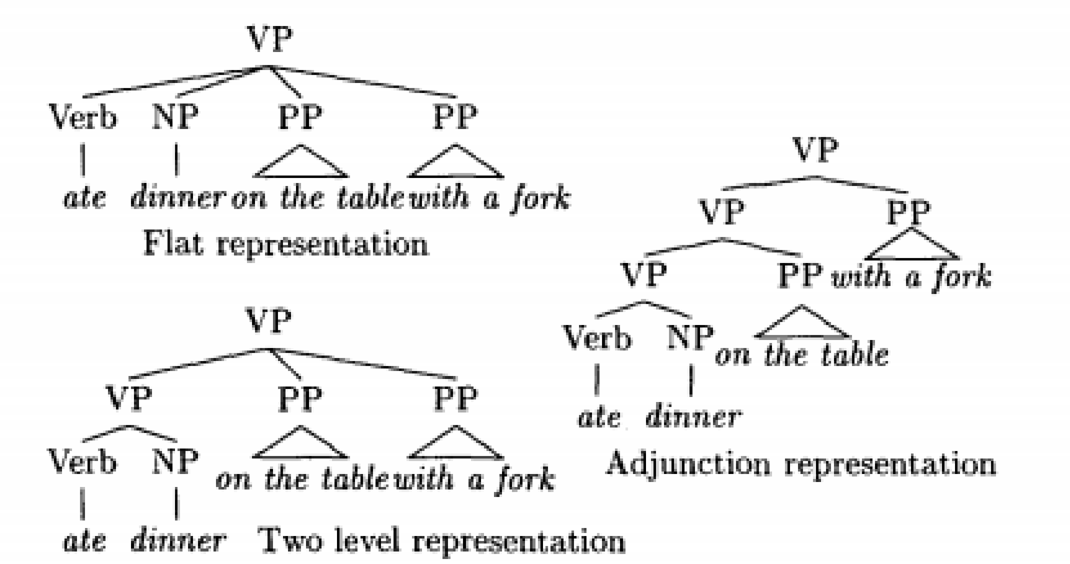


Figure 11.6: The left parse is preferable because of the conjunction of phrases headed by *France* and *Italy*.

Subsumption There are several choices for annotating PP attachment



Johnson (1998) shows that even though the two-level representation is chosen in the annotation, it can never be produced by a PCFG because the production is

subsumed.

$$P(\text{NP} \rightarrow \text{NP PP}) = 0.112 \tag{11.9}$$

$$P(\text{NP} \rightarrow \text{NP PP PP}) = 0.006 \tag{11.10}$$

$$P(\text{NP} \rightarrow \text{NP PP})P(\text{NP} \rightarrow \text{NP PP}) = (0.112)^2 \approx 0.013 \tag{11.11}$$

The probability of applying the NP → NP PP production twice is greater than the probability of the two-PP production, so this production will never appear in a PCFG parse. Johnson shows that 9% of all productions are subsumed and can be removed from the grammar!

Modern generative parsing algorithms improve on pure PCFG parsing by automatically refining the non-terminals. There are three main ways to do this:

Tree transformations The annotated parse trees are automatically transformed so that the production probabilities are more useful for automatic parsing.

Lexicalization Each non-terminal is labeled with a **head word**, indicating the most syntactically important word in the constituent that the non-terminal derives.

Unsupervised machine learning The original non-terminal set is automatically refined into more precise categories that make PCFG parsing easier. One way to do this is by **expectation-maximization** (chapter 4).

The first two approaches are discussed in the remainder of this section; non-terminal refinement is discussed in section 11.6.

Tree transformations

Johnson (1998) proposed a series of heuristic transformations to the Penn Treebank annotations. At training time, he applies these transformations to the training data, and learn the probabilities of the PCFG productions. This parser is then applied to the test data. The resulting parses must then be **detransformed** so that they can be evaluated against the original ground truth.

Flattening The first transformation is to "flatten" nested noun phrases to be more like verb phrase structures, as shown in Figure 11.7.

Flattened rules are of course still context-free, but by reducing recursion, they allow more specific probabilities to be learned. This can eliminate the problems with rule subsumption that we saw earlier.

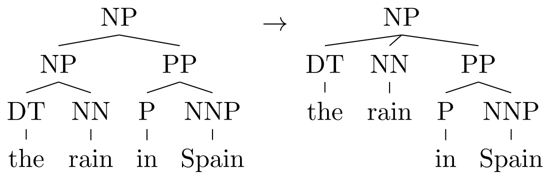


Figure 11.7: Johnson (1998) "flattens" nested noun phrases to remove internal structure.[todo: bigger arrow]

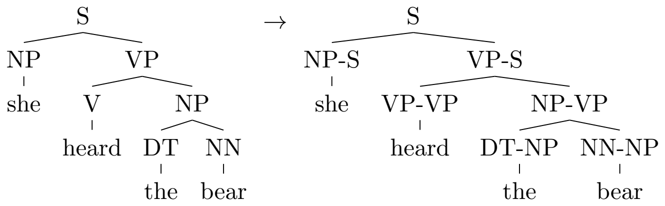


Figure 11.8: Parent annotation in a CFG derivation

Parent annotation Context-free grammars assume that the probability of each production depends only on the identity of the non-terminal on the left-hand side, and not on anything else in the derivation. But in PTB-style analysis of English grammar, the observed probability of productions often depends on the parent of the element on the left-hand side. For example, in the PTB, noun phrases are much more likely to be modified by prepositional phrases when they are in the object position (e.g., *They amused the students from Georgia*) than in the subject position (e.g., *The students from Georgia amused them*). In PCFG terms, this means that the NP → NP PP production is more likely if the entire consistuent is the child of a VP than if it is the child of S.

$$\begin{aligned} P(\text{NP} \rightarrow \text{NP PP}) =& 11\% \qquad (11.12)\\ P(\text{NP} \ \textsc{under} \ \text{S} \rightarrow \text{NP PP}) =& 9\% \qquad (11.13)\\ P(\text{NP} \ \textsc{under} \ \text{VP} \rightarrow \text{NP PP}) =& 23\% \qquad (11.14) \end{aligned}$$

We can capture this phenomenon via **parent annotation**: augmenting each non-terminal with the identity of its parent (Figure 11.8). This is sometimes called **vertical Markovization**, since we introduce a Markov dependency between each node and its parent (Klein and Manning, 2003).

Parent annotation weakens the PCFG independence assumptions. This could help accuracy by making more fine-grained distinctions, which better capture real lingusitic phenomena. But it could also hurt accuracy, because each production probability must be estimated from less data.

In practice, the transformations proposed by Johnson (1998) do improve performance on PTB parsing:

- Standard PCFG: 72% F-measure, 14,962 rules
- Parent-annotated PCFG with flattening: 80% F-measure, 22,773 rules [todo: double check that flattening is included too]
- In principle, parent annotation could have increased the grammar size much more dramatically, but many possible productions never occur, or are subsumed.

Lexicalization

Recall that some of the problems with PCFG parsing that were suggested above have to do with **meaning** — for example, preferring to coordinate constituents that are of the same type, like *cats* and *dogs* rather than *cats* and *houses*. A simple way to capture semantics is through the words themselves: we can annotate each non-terminal with **head** word of the phrase.

Head words are deterministically assigned according to a set of rules, sometimes called **head percolation rules**. In many cases, these rules are straightforward: the head of a NP → DT N production is the noun, the head of a S → NP VP production is the head of the VP, etc. But as always, there are a lot of special cases.

A fragment of the head percolation rules used in many parsing systems are found in Table 11.3.[4]

Non-terminal	Direction	Priority
S	right	VP SBAR ADJP UCP NP
VP	left	VBD VBN MD VBZ TO VB VP VBG VBP ADJP NP
NP	right	N* EX $ CD QP PRP . . .
PP	left	IN TO FW

Table 11.3: A fragment of head percolation rules

The meaning of these rules is that to find the head of an S constituent, we first look for the rightmost VP child; if we don't find one, we look for the rightmost

[4] From `http://www.cs.columbia.edu/~mcollins/papers/heads`

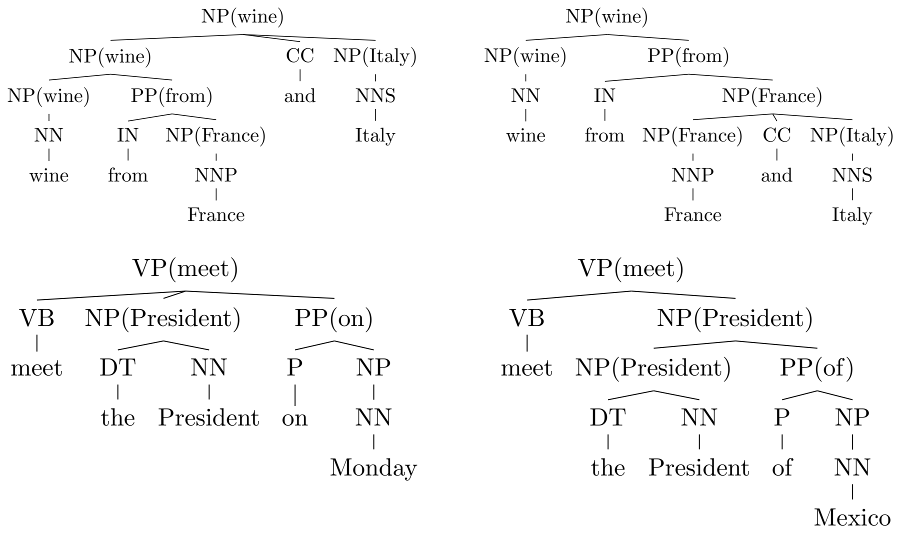


Figure 11.9: Lexicalization can address ambiguity on coordination scope (upper) and PP attachment (lower)

SBAR child, and so on down the list. Verb phrases are headed by left verbs (the head of *can walk home* is *walk*, since *can* is tagged MD), noun phrases are headed by the rightmost noun-like non-terminal (so the head of *the red cat* is *cat*), and prepositional phrases are headed by the preposition (the head of *at Georgia Tech* is *at*). Some of these rules are somewhat arbitrary — there's no particular reason why the head of *cats and dogs* should be *dogs* — but the point here is just to get some lexical information that can support parsing, not to make any deep claims about syntax.

Given these rules, we can lexicalize the parse trees for some of our examples, as shown in Figure 11.9.

- In the upper part of Figure 11.9, we see how lexicalization can help solve coordination scope ambiguity; if,

  $$P(\text{NP} \to \text{NP}(\textit{France})\ \text{CC}\ \text{NP}(\textit{Italy})) > P(\text{NP} \to \text{NP}(\textit{wine})\ \text{CC}\ \text{NP}(\textit{Italy})), \tag{11.15}$$

  we should get the right parse.

- In the lower part of Figure 11.9, we see how lexicalization can help solve attachment ambiguity. Here we assume that,

$$P(\text{VP}(\textit{meet}) \to \alpha \; \text{PP}(\textit{on})) \gg P(\text{NP}(\textit{President}) \to \beta \; \text{PP}(\textit{on})) \tag{11.16}$$

$$P(\text{VP}(\textit{meet}) \to \alpha \; \text{PP}(\textit{of})) \ll P(\text{NP}(\textit{President}) \to \beta \; \text{PP}(\textit{of})) \tag{11.17}$$

In plain English: *Meeting* are *on* things; *Presidents* are *of* things.

- Recall that verbs may be intransitive, transitive, or ditransitive. Lexicalization can help distinguish these cases, as shown by the lexicalized PCFG probabilities for the ditransitive VP production,

$$P(\text{VP} \to \text{V NP NP}) = 0.00151 \tag{11.18}$$

$$P(\text{VP}(\textit{said}) \to \text{V}(\textit{said}) \text{ NP NP}) = 0.00001 \tag{11.19}$$

$$P(\text{VP}(\textit{gave}) \to \text{V}(\textit{gave}) \text{ NP NP}) = 0.01980. \tag{11.20}$$

Overall, lexicalization had a major impact on parsing accuracy, as shown in Table 11.4. According to Eugene Charniak, one of the early proponents of lexicalized PCFG parsing: "To do better, it is necessary to condition probabilities on the actual words of the sentence. This makes the probabilities much tighter."[5]

Vanilla PCFG	72%
Head-annotated PCFG (Johnson, 1998)	80%
Lexicalized PCFGs (Collins, 1997, 2003; Charniak, 1997)	87-89%

Table 11.4: Penn Treebank parsing accuracies

Algorithms for lexicalized parsing

In principle, we could perform lexicalized PCFG parsing with the CKY algorithm, by expanding the non-terminals to include the cross-product of all PTB non-terminals and all words. Then our grammar would include rules like:

$$\text{V}\textsc{p}(\textit{scratch}) \to \text{V}\textsc{p}(\textit{scratch})\text{N}\textsc{p}(\textit{people}) \tag{11.21}$$

$$\text{V}\textsc{p}(\textit{scratch}) \to \text{V}\textsc{p}(\textit{scratch})\text{N}\textsc{p}(\textit{themselves}) \tag{11.22}$$

$$\text{V}\textsc{p}(\textit{scratch}) \to \text{V}\textsc{p}(\textit{scratch})\text{N}\textsc{p}(\textit{Abigail}) \tag{11.23}$$

$$\ldots\ldots \tag{11.24}$$

[5] The quote is from a workshop at Johns Hopkins University in 2000.

In a sense, we have gone from N non-terminals (S, VP, ...) to $N \times V$ non-terminals ($\text{S}(\textit{scratch}), \text{S}(\textit{eat}), \text{VP}(\textit{scratch}), \text{VP}(\textit{eat}), \ldots$). This would imply $\mathcal{O}(N^3V^3)$ possible productions. Since one of the two children must have the same head word as the parent, the situation is slightly better: $\mathcal{O}(N^3V^2)$. But since the vocabulary size is at least 10^4 in most reasonable scenarios, this is still not practical.

With a little thought, it should be clear that the complexity need not depend on V. All the words are already given, so the only question is which word **position** in $h \in \{1 \ldots M\}$ is the head of each non-terminal, and not which word type $w \in \mathcal{V}$ is the head. We can implement this intuition by modifying the CKY algorithm, building a different chart structure. We will still work bottom-up, but now we need one additional piece of information: the location of the head word of each span. We should therefore store the elements $t[i,j,h,X]$, indicating a span over the substring $\boldsymbol{w}_{i:j-1}$, headed by w_h ($h \in i \ldots j-1$), with parent node X.

To recursively construct $t[i,j,h,X]$, we need to consider two possibilities: either the head h is in the left child, or it is in the right child. If h is in the left child, then the split point k must be greater than h. Finally, in addition to maximizing over the location of the split point, we must also maximize over locations of the head of the right child, $\ell \geq k$. We can then compute $t_\ell[i,j,h,X]$, which is the score of the best derivation $X(w_h) \rightsquigarrow \boldsymbol{w}_{i,j}$ in which the head word w_h is in the left child:

$$t_\ell[i,j,h,X] = \max_{k>h} \max_{k \leq \ell < j} \max_{\text{X}(w_h) \to \text{Y}(w_h)\text{Z}(w_\ell)} P(\text{X}(w_h) \to \text{Y}(w_h)\text{Z}(w_m)) \times t[i,k,h,Y] \times t[k,\ell,j,Z] \tag{11.25}$$

If the head h is in the right child, then the split point k must be less than or equal to h. We must also identify the location of the head of the left child, $\ell < k$. We can then compute $t_r[i,j,h,X]$, which is the score of the best derivation $X(w_h) \rightsquigarrow \boldsymbol{w}_{i,j}$ in which the head word w_h is in the right child:

$$t_r[i,j,h,X] = \max_{k \leq h} \max_{i \leq \ell < k} \max_{\text{X}(w_h) \to \text{Y}(w_\ell)\text{Z}(w_h)} P(\text{X}(w_h) \to \text{Y}(w_\ell)\text{Z}(w_h)) \times t[i,k,\ell,Y] \times t[k,j,h,Z]. \tag{11.26}$$

Finally, we can compute the score of the overall best derivation $X(w_h) \rightsquigarrow \boldsymbol{w}_{i,j}$ as the max of the scores of the best left-headed and right-headed derivations,

$$t[i,j,h,X] = \max\left(t_\ell[i,j,h,X], t_r[i,j,h,X]\right). \tag{11.27}$$

In this headed version of CKY, we are building a table of size $\mathcal{O}(M^3N)$, where M is the length of the sentence and N is the number of non-terminals. To fill in each cell, we must perform $\mathcal{O}(M^2G)$ operations, taking maxes over two indices

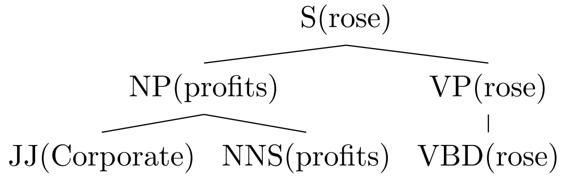


Figure 11.10: Example of a lexicalized derivation for the CHarniak parser

in the sentence, and over all rules. This would imply a total time complexity of $\mathcal{O}(M^5NG)$ — still too slow to be practical, even without the dependency on the vocabulary size V. However, Eisner and Satta (1999) show that a more clever algorithm reduces this time cost back to $\mathcal{O}(M^3G)$. A more serious problem is **estimation**: all this work on parsing algorithms doesn't save us from computing probabilities for all $\mathcal{O}(N^3V^2)$ possible productions. Charniak (1997) and Collins (1997, 2003) offer practical solutions, which decompose the production probabilities using various independence assumptions.

The Charniak Parser

The Charniak (1997) parser gives a relatively straightforward way to lexicalize PCFGs. Head probabilities capture "bilexical" phenomena; in the example, ... *meet the President of Mexico*, the bilexical probabilities for the pairs $\langle \textit{meet}, \textit{of} \rangle$ and $\langle \textit{President}, \textit{of} \rangle$ should help the parser make the right attachment decision. We can capture this idea by representing the probability of each production $\text{X}(i) \to \text{Y}(j)\text{Z}(k)$, by the product of two factors:

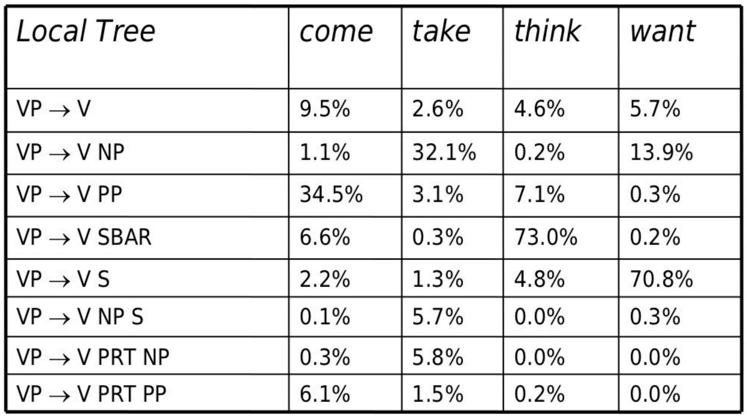
- The **rule probability**, $P(r \mid w_m, t_m, t_{\rho(m)})$, where r is the rule X → Y Z, m is the index of the head of the left-hand side, t_m is the type of the left-hand side (a non-terminal, such as VP), $t_{\rho(m)}$ is the type of the parent of m (again, a non-terminal).
- The **head probability**, $P(w_m \mid w_{\rho(m)}, t_m, t_{\rho(m)})$, where w_m is a head word.

Consider the example in Figure 11.10. The rule probability for the noun phrase production is,

$$P(\text{NP} \to \text{JJ NNS} \mid w_m = \textit{rose}, t_m = \text{NP}, t_{\rho(m)} = \text{S}). \tag{11.28}$$

The head probability is,

$$\text{p}(\textit{profits} \mid w_{\rho(m)} = \textit{rose}, t_m = \text{NP}, t_{\rho(m)} = \text{S}). \tag{11.29}$$



Local Tree	*come*	*take*	*think*	*want*
VP → V	9.5%	2.6%	4.6%	5.7%
VP → V NP	1.1%	32.1%	0.2%	13.9%
VP → V PP	34.5%	3.1%	7.1%	0.3%
VP → V SBAR	6.6%	0.3%	73.0%	0.2%
VP → V S	2.2%	1.3%	4.8%	70.8%
VP → V NP S	0.1%	5.7%	0.0%	0.3%
VP → V PRT NP	0.3%	5.8%	0.0%	0.0%
VP → V PRT PP	6.1%	1.5%	0.2%	0.0%

Figure 11.11: The probability of verb phrase complements is highly dependent on the identity of the verb itself: for example, the verb *come* frequently takes a prepositional phrase as a complement (*come to the party*), while the verb *take* is more likely to take a noun phrase complement. Conditioning on the verb identity can therefore improve parsing accuracy. [todo: attribution for this table]

We would then multiply these probabilities to fill in the chart,

$$
\begin{aligned}
t[1,3,2,\text{NP}] =& P(\text{NP} \to \text{JJ NNS} \mid w_m = \textit{rose}, t_m = \text{NP}, t_{\rho(m)} = \text{S}) && (11.30)\\
&\times \text{p}(\textit{profits} \mid w_{\rho(m)} = \textit{rose}, t_m = \text{NP}, t_{\rho(m)} = \text{S}). && (11.31)
\end{aligned}
$$

Bilexical probabilities are captured in the head probability, which depends on the head words of both the parent and child. This parser therefore combines two ideas that we have seen before:

Head annotation since both the rule and head probabilities depend on the parent type $t_{\rho(m)}$.

Lexicalization since the rule probability depends on the head word w_m. These rule probabilities can capture phenomena like verb complement frames, as shown in Figure 11.11.

Estimating the Charniak parser The Charniak parser involves fewer parameters than a naive lexicalized PCFG. To estimate the relevant parameters in our example, we have

$$\begin{aligned}
&\mathrm{p}_{\text{head}}(\textit{profits} \mid t_m = \text{NP}, t_{\rho(m)} = \text{S}, w_{\rho(m)} = \textit{rose}) \\
&\quad = \frac{\text{count}(w_m = \textit{profits}, t_m = \text{NP}, t_{\rho(m)} = \text{S}, w_{\rho(m)} = \textit{rose})}{\text{count}(t_m = \text{NP}, t_{\rho(m)} = \text{S}, w_{\rho(m)} = \textit{rose})} \\
&P_{\text{rule}}(\text{NP} \to \text{JJ NNS} \mid w_{\rho(m)} = \textit{rose}, t_m = \text{NP}, t_{\rho(m)} = \text{S}) \\
&\quad = \frac{\text{count}(\text{NP} \to \text{JJ NNS}, t_m = \text{NP}, t_{\rho(m)} = \text{S}, w_{\rho(m)} = \textit{rose})}{\text{count}(t_m = \text{NP}, t_{\rho(m)} = \text{S}, w_{\rho(m)} = \textit{rose})}
\end{aligned}$$

The Penn Treebank provides is still the main dataset for syntactic analysis of English. Yet its 1M words is not nearly enough data to accurately estimate lexicalized models such as the Charniak parser, without smoothing. For example, in 965K annotated constituent spans, there are only 66 examples of WHADJP, and only 6 of these aren't *how much* or *how many*.[todo: cite?]

In the example above (*corporate profits rose*), the unsmoothed head probability is zero, as estimated from the PTB: there are zero counts of *profits* headed by *rose* in the treebank [todo: check]. In general, bilexical counts are going to be very sparse. But the "backed-off" probabilities give a reasonable approximation. These can be incorporated via interpolation.

Smoothing the Charniak Parser We compute a smoothed estimate of the head probability as,

$$\begin{aligned}
\hat{\mathrm{p}}(w_m \mid t_m, w_{\rho(m)}, t_{\rho(m)}) = &\lambda_1 \mathrm{p}_{mle}(w_m \mid t_m, w_{\rho(m)}, t_{\rho(m)}) \\
&+ \lambda_2 \mathrm{p}_{mle}(w_m \mid t_m, \text{cluster}(w_{\rho(m)}), t_{\rho(m)}) \\
&+ \lambda_3 \mathrm{p}_{mle}(w_m \mid t_m, t_{\rho(m)}) \\
&+ \lambda_4 \mathrm{p}_{mle}(w_m \mid t_m),
\end{aligned} \tag{11.32}$$

where cluster$(w_{rho(m)})$ is the cluster of word $w_{\rho(m)}$, obtained by applying an automatic clustering method to **distributional** statistics (Pereira et al., 1993); see chapter 15 for more details.

For example:

	$\mathrm{p}(\textit{profit} \mid NP, \textit{rose}, S)$	$P(\textit{corp.} \mid JJ, \textit{profit}, NP)$
$\mathrm{p}(w_m \mid t_m, w_{\rho(m)}, t_{\rho(m)})$	0	.245
$\mathrm{p}(w_m \mid t_m, c(w_{\rho(m)}), t_{\rho(m)})$	.0035	.015
$\mathrm{p}(w_m \mid t_m, t_{\rho(m)})$	.00063	.0053
$\mathrm{p}(w_m \mid t_m)$	.00056	.0042

We have to tune $\lambda_1 \ldots \lambda_4$, and an equivalent set of parameters for the rule probabilities.

The Charniak parser suffers from acute sparsity problems because it estimates the probability of entire rules. Another extreme would be to generate the children independently from each other, e.g.

$$P(\text{S} \rightarrow \text{NP VP}) \approx P_L(\text{S} \rightarrow \text{NP}) P_R(\text{S} \rightarrow \text{VP}) \tag{11.33}$$

Collins (2003) and Charniak (2000) make a compromise: their parsers estimate lexicalized probabilities that condition on the parent and the head child.

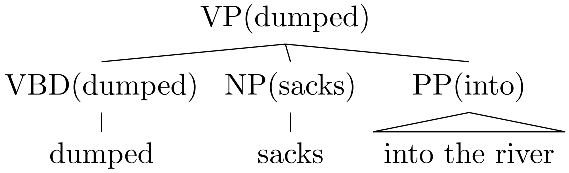
The Collins Parser

The Charniak parser focuses on lexical relationships between children and parents. Motivated by the linguistic theory of **lexicalized tree-adjoining grammar** (Joshi and Schabes, 1997), the Collins (2003) parser focuses on relationships between adjacent children of the same parent. We can write each production as,

$$X \rightarrow L_m L_{m-1} \ldots L_1 H R_1 \ldots R_{n-1} R_n,$$

where H is the child containing the head word, each L_i is a child element to the left of the head, and each R_j is a child element to the right of the head. In the Collins parser, these elements are generated probabilistically from the head outward. The outermost elements of L and R are special ♦ symbols.

For example, consider the verb phrase,



To model this rule, we would compute:

$$\text{p}(\text{VP}(\textit{dumped}, \text{VBD}) \rightarrow [\blacklozenge, \text{VBD}(\textit{dumped}, \text{VBD}), \text{NP}(\textit{sacks}, \text{NNS}), \text{PP}(\textit{into}, \text{P}), \blacklozenge])$$

We compute this probability through a hypothesized generative process,

- Generate the head:

$$P(H \mid LHS) = P(\text{VBD}(\textit{dumped}, \text{VBD}) \mid \text{VP}(\textit{dumped}, \text{VBD})) \tag{11.34}$$

- Generate the left dependent:

$$P_L(\blacklozenge \mid \text{VP}(\textit{dumped}, \text{VBD}), \text{VBD}(\textit{dumped}, \text{VBD})) \tag{11.35}$$

- Generate the right dependent:

$$P_R(\text{NP}(\textit{sacks}, \text{NNS}) \mid \text{VP}(\textit{dumped}, \text{VBD}), \text{VBD}(\textit{dumped}, \text{VBD})) \tag{11.36}$$

- Generate the right dependent:

$$P_R(\text{NP}(into, PP) \mid \text{VP}(\textit{dumped}, \text{VBD}), \text{VBD}(\textit{dumped}, \text{VBD})) \tag{11.37}$$

- Generate the right dependent:

$$P_R(\blacklozenge \mid \text{VP}(\textit{dumped}, \text{VBD}), \text{VBD}(\textit{dumped}, \text{VBD})) \tag{11.38}$$

The rule probability is the product of these generative probabilities. Because these generative probabilities are defined only over parts of the productions, they are easier to estimate from limited data. Nonetheless, it is still necessary to smooth these probabilities by interpolating them with less expressive probability functions. For example,

$$\begin{aligned}&\hat{P}_R(\text{NP}(\textit{sacks}, \text{NNS}) \mid \text{VP}(\textit{dumped}, \text{VBD}), \textit{dumped}, \text{VBD})\\ =&\lambda_1 \hat{P}(\text{NP}(\textit{sacks}, \text{NNS}) \mid \text{VP}, \textit{dumped}, \text{VBD})\\ &+ \lambda_2 \hat{P}(\text{NP}(\textit{sacks}, \text{NNS}) \mid \text{VP}, \text{VBD})\\ &+ \lambda_3 \hat{P}(\text{NP}(\textit{sacks}, \text{NNS}) \mid \text{VP})\end{aligned} \tag{11.39}$$

The Collins parser models bilexical dependencies between the head and its siblings. Bilexical probabilities require counts over pairs of words, a space of $\mathcal{O}(V^2)$ events. It is this large event space that makes these probabilities difficult to estimate, necessitating smoothing. Is it worth it? Bikel (2004) evaluating the importance of bilexical probabilities to the performance of the Collins parser. In general, these bilexical probabilites are rarely available — because most of the possible bilexical pairs in the test data are unobserved in the training data — but these bilexical probabilities are indeed active in 29% of the rules in the **top-scoring** parses. Still, Bikel finds that bilexical probabilities play a relatively small role in accuracy: an equivalent parser which conditions on only a single head suffers only 0.3% decrease in F-measure. A completely unlexicalized parser performs considerably worse, indicating that some amount of lexicalization is still necessary for top performance.

Summary of lexicalized parsing

Lexicalized parsing results in substantial accuracy gains:

Vanilla PCFG	72%
Parent-annotations (Johnson, 1998)	80%
(Charniak, 1997)	86%
(Collins, 2003)	87%

Table 11.5: Accuracies for lexicalized parsers

But lexicalization creates an explosion in the size of the grammar, which requires elaborate smoothing techniques and makes parsing slow. Treebank syntactic categories are too coarse, but lexicalized categories may be too fine; more recent approaches have sought middle ground. At the same time, natural language processing has moved from generative models to more advanced machine learning techniques in the late 1990s and early 2000s, and researchers have worked to incorporate these techniques into parsing. We consider both of these ideas in the next section.

11.6 Modern constituent parsing

Reranking

Charniak and Johnson (2005) and Collins and Koo (2005) combine generative and discriminative models for parsing, using the idea of **reranking**. First, a generative model is used to identify its K-best parses. Then a discriminative ranker is trained to select the best of these parses. The discriminative model does not need to search over all parses — just the best K identified by the generative model. This means that it can use arbitrary features — such as structural features that capture parallelism and right-branching, which could not be easily incorporated into a bottom-up parsing model. Because learning is discriminative, rerankers can also use very rich lexicalized feature spaces, relying on regularization to combat overfitting. Overall, this approach yields substantial improvements in accuracy on the Penn Treebank, and can be applied to improve any generative parsing model.

Refinement grammars

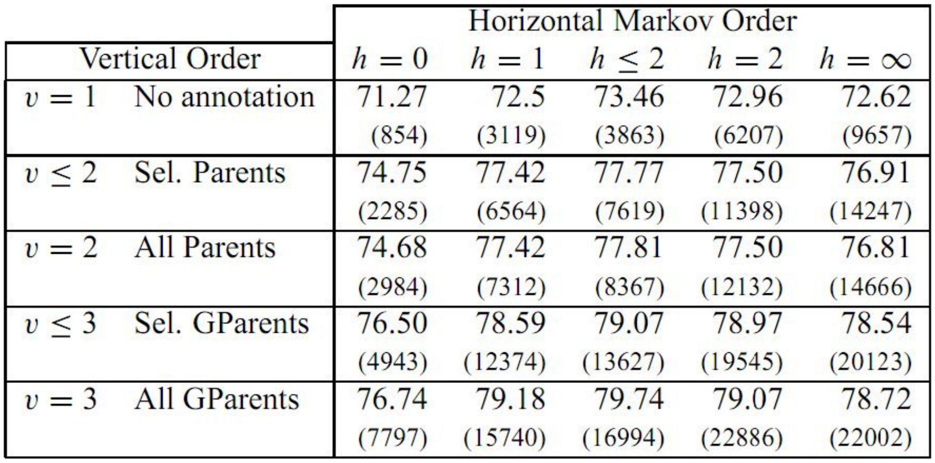
Klein and Manning (2003) revisit unlexicalized parsing, expanding on the ideas in

(Johnson, 1998).

They apply two types of **Markovization**:

- Vertical Markovization, making the probability of each parsing rule depend not only on the type of the parent symbol, but also on its parent type. This is identical to the parent annotation proposed by Johnson (1998). The amount of vertical Markovization can be written v, with $v = 1$ indicating a standard PCFG.
- Horizontal Markovization, where the probability of each child depends on only some of its siblings. In a standard PCFG $h = \infty$, since there is no decomposition on the right-hand side of the rule. In the Collins parser, different settings of h were explored, with $h = 1$ indicating dependence only on the head, and $h = 2$ indicating dependence on the nearest sibling as well as the head.

A comparison of various Markovization parameters is shown in Figure 11.12:



		Horizontal Markov Order				
Vertical Order		$h = 0$	$h = 1$	$h \leq 2$	$h = 2$	$h = \infty$
$v = 1$	No annotation	71.27	72.5	73.46	72.96	72.62
		(854)	(3119)	(3863)	(6207)	(9657)
$v \leq 2$	Sel. Parents	74.75	77.42	77.77	77.50	76.91
		(2285)	(6564)	(7619)	(11398)	(14247)
$v = 2$	All Parents	74.68	77.42	77.81	77.50	76.81
		(2984)	(7312)	(8367)	(12132)	(14666)
$v \leq 3$	Sel. GParents	76.50	78.59	79.07	78.97	78.54
		(4943)	(12374)	(13627)	(19545)	(20123)
$v = 3$	All GParents	76.74	79.18	79.74	79.07	78.72
		(7797)	(15740)	(16994)	(22886)	(22002)

Figure 11.12: Performance for various Markovization levels (Klein and Manning, 2003).

Second, Klein and Manning note that the right level of linguistic detail is somewhere between treebank categories and individual words. For example:

- Some parts-of-speech and non-terminals are truly substitutable: for example, *cat*/N and *dog*/N.

- But others are not: for example, *on*/PP behaves differently from *of*/PP. This is an example of **subcategorization**.
- Similarly, the words *and* and *but* should be distinguished from other coordinating conjunctions.

Figure 11.13 shows an example of an error that is corrected through the introduction of a new NP-TMP subcategory for temporal noun phrases.

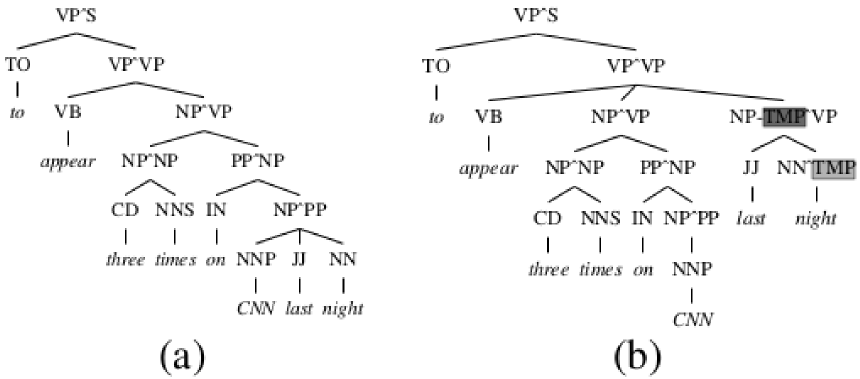


Figure 11.13: State-splitting creates a new non-terminal called NP-TMP, for temporal noun phrases. This corrects the PCFG parsing error in (a), resulting in the correct parse in (b).

Automated state-splitting Klein and Manning (2003) use linguistic insight and error analysis to manually split PTB non-terminals so as to make parsing easier. Later work by Dan Klein and his students automated this state-splitting process, by treating the "refined" non-terminals as latent variables. For example, we might split the noun phrase non-terminal into NP1, NP2, NP3, . . . , without defining in advance what each refined non-terminal corresponds to.

Petrov et al. (2006) employ expectation-maximization to solve this problem. In the E-step, we estimate a **marginal** distribution q over the refinement type of each non-terminal. Note that this E-step is subject to the constraints of the original Penn Treebank annotation: an NP can be reannotated as NP4, but not as VP3. Now, the marginals are defined as $\mathrm{p}(X \rightsquigarrow \boldsymbol{w}_{i:j} \mid \boldsymbol{w}_{1:M})$, which is the probability that the span $i : j$ is derived from X, conditioning on the entire sentence $\boldsymbol{w}_{1:M}$ and marginalizing over all other parts of the derivation. In the forward-backward algorithm, we computed similar marginals for sequence labeling. In the context of context-free

Proper nouns			
NNP-14	*Oct.*	*Nov.*	*Sept.*
NNP-12	*John*	*Robert*	*James*
NNP-2	*J.*	*E.*	*L.*
NNP-1	*Bush*	*Noriega*	*Peters*
NNP-15	*New*	*San*	*Wall*
NNP-3	*York*	*Francisco*	*Street*
Personal Pronouns			
PRP-0	*It*	*He*	*I*
PRP-1	*it*	*he*	*they*
PRP-2	*it*	*them*	*him*

Table 11.6: Examples of automatically refined non-terminals and some of the words that they generate (Petrov et al., 2006).

grammars, the corresponding algorithm is called **inside-outside** (Lari and Young, 1990): each marginal is computed as a product of an **inside probability** defined in section 11.3), and an **outside probability**, which is computed recursively from the top down.

In the M-step, we recompute the parameters of the grammar, based on the expected counts from the E-step. As usual, this process can be iterated to convergence. To determine the number of refinement types for each tag, Petrov et al. (2006) apply a split-merge heuristic; Liang et al. (2007) and Finkel et al. (2007) apply Bayesian nonparametrics.

This approach yielded state-of-the-art accuracy at the time, with an F-measure of 90.6%. Some examples of refined non-terminals are shown in Table 11.6. The proper nouns differentiate months, first names, middle initials, last names, first names of places, and second names of places; each of these will tend to appear in different parts of grammatical productions. The personal pronouns differentiate grammatical role, with PRP-0 appearing in subject position at the beginning of the sentence (note the capitalization), PRP-1 appearing in subject position but not at the beginning of the sentence, and PRP-2 appearing in object position.

Discriminative parsing

In sequence labeling, discriminative models such as structured perceptron and conditional random field did much better than the generative hidden Markov model. We can think of a PCFG parser in our usual framework of structured prediction:

$$\hat{\tau} = \text{argmax}\, \tau \boldsymbol{\theta}^\top \boldsymbol{f}(\tau, \boldsymbol{w}). \tag{11.40}$$

In this case, the features $\boldsymbol{f}(\tau, \boldsymbol{w})$ count all the CFG productions in τ and the terminal productions to $\boldsymbol{w}$, and the weights $\boldsymbol{\theta}$ count the log-probabilities of those productions.[todo: explain in more detail how this would work with CKY]

This suggests that we could try to learn the weights $\boldsymbol{\theta}$ discriminatively. But if we are willing to learn the weights discriminatively, we can also add additional features; we only require a feature decomposition so that $\boldsymbol{f}(\tau, \boldsymbol{w})$ decomposes across the productions in τ, so that we can still perform CKY parsing to find the best-scoring parse. For example, under such a decomposition, we could incorporate lexical features, so that we learn weights for the non-terminal production as well as for lexicalized forms,

$f1$ NP($*$) $\rightarrow$ NP(*) PP(*)

$f2$ NP($\textit{cats}$) $\rightarrow$ NP($\textit{cats}$) PP($*$)

$f3$ NP($*$) $\rightarrow$ NP($*$) PP($\textit{claws}$)

$f4$ NP($\textit{cats}$) $\rightarrow$ NP($\textit{cats}$) PP($\textit{claws}$)

Through regularization, we can find weights that strike a good balance between frequently-observed features ($f1$) and more discriminative features ($f4$).

This approach was implemented by Finkel et al. (2008) in the context of PCFG parsing with Conditional Random Fields. They used stochastic gradient descent for training, with the inside-outside algorithm (analogous to forward-backward, but for trees) to compute expected feature counts. However, the time complexity of $\mathcal{O}(M^3)$ posed serious challenges — recall that CRF sequence labeling can be trained in linear time. Finkel et al. (2008) address these issues by "prefiltering" the CKY parsing chart, identifying the productions which cannot be part of any complete parse.

Carreras et al. (2008) use the averaged perceptron to perform conditional parsing, employing an alternative feature decomposition based on tree-adjoining grammar (TAG). This yields substantially better results, at $F = 90.5$.

Other parsing models

Table 11.7 summarizes a number of results on parsing. Since the observations of Johnson (1998) about the poor performance of straightforward PCFG parsing, the error rate has been reduced from 28% to 8-9% — more than a three-fold error

Vanilla PCFG	72%
Parent-annotations (Johnson, 1998)	80%
Lexicalized (Charniak, 1997)	86%
Lexicalized (Collins, 2003)	87%
Lexicalized, reranking, self-training (McClosky et al., 2006)	92.1%
State splitting (Petrov and Klein, 2007)	90.1%
CRF Parsing (Finkel et al., 2008)	89%
TAG Perceptron Parsing (Carreras et al., 2008)	91.1%
Compositional Vector Grammars (Socher et al., 2013a)	90.4%
Neural CRF (Durrett and Klein, 2015)	91.1%

Table 11.7: Penn Treebank parsing scoreboard, circa 2015 (Durrett and Klein, 2015)

reduction. One notable alternative not described in detail here is the **self-training** parser of McClosky et al. (2006), which automatically labels additional training instances, and then uses them for learning. Self-training is often considered to be a risky technique in machine learning, since the automatically-labeled instances can cause the classifier to "drift" away from the correct model (Blum and Mitchell, 1998).

Recent work has applied neural representations to parsing, representing units of text with dense numerical vectors (Socher et al., 2013a; Durrett and Klein, 2015). Neural approahes to natural language processing will be surveyed in chapter 21. For now, we note that while performance for these models is at or near the state-of-the-art, neural net architectures have not demonstrated the same dramatic improvements in natural language parsing as in other problem domains, such as computer vision (e.g., Krizhevsky et al., 2012).

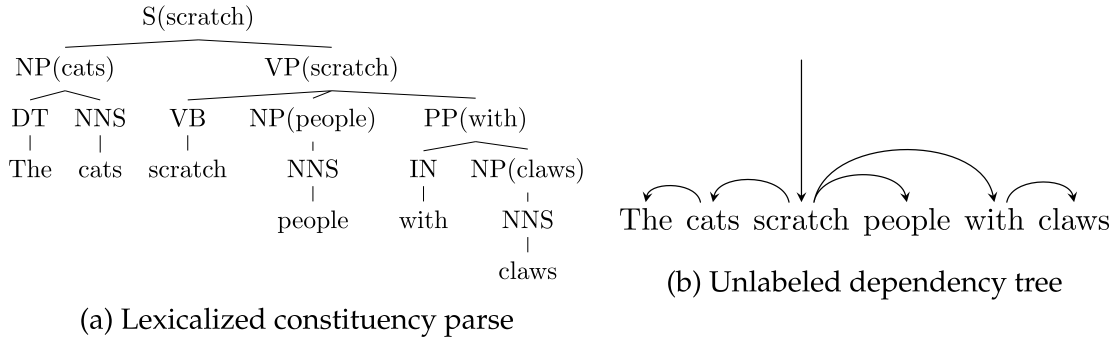
Chapter 12

Dependency Parsing

The previous chapter discussed algorithms for analyzing sentences in terms of nested **constituents**, such as noun phrases and verb phrases. The combination of constituency structure and head-percolation rules yields a set of **dependencies** between individual words. These dependencies are a more "bare-bones" version of syntax, leaving out information that is present in the full constituent parse. Nonetheless, the dependency representation is still capable of capturing important linguistic phenomena, such as the prepositional phrase attachment and coordination scope. For this reason, dependency parsing is increasingly used in applications that require syntactic analysis. While dependency structures can be obtained as a byproduct of constituent parsing, it is more efficient to extract them directly. Indeed, accurate dependency parses can be obtained by algorithms with time complexity that is linear in the length of the sentence. This chapter begins by overviewing dependency grammar, and then presents the two dominant approaches to dependency parsing, graph-based and transition-based dependency parsing.

12.1 Dependency grammar

In lexicalized parsing, non-terminals such as NP are augmented with **head words**, as shown in Figure 12.1a. In this sentence, the head of the S constituent is the main verb, *scratch*; this non-terminal then produces the noun phrase *the cats*, whose head word is *cats*, and from which we finally derive the word *the*. Thus, the word *scratch* occupies the central position for the sentence, with the word *cats* playing a supporting role. In turn, *cats* occupies the central position for the noun phrase, with the word *the* playing a supporting role.



(a) Lexicalized constituency parse

(b) Unlabeled dependency tree

Figure 12.1: Dependency grammar is closely linked to lexicalized context free grammars: each lexical head has a dependency path to every other word in the constituent.

These relationships, which hold between the words in the sentence, can be formalized in a directed graph structure. In this graph, there is an edge from word i to word j iff word i is the head of the first branching node above a node headed by j. Thus, in our example, we would have *scratch* → *cats* and *cats* → *the*. We would not have the edge *scratch* → *the*, because although *cats* dominates *the* in the graph, *cats* is not the head of a node that produces a node headed by *the*. These edges describe syntactic **dependencies**, a bilexical relationship between a **head** and a **dependent**, which is at the heart of **dependency grammar** (Tesnière, 1966).

If we continue to build out this **dependency graph**, we will eventually reach every word in the sentence, as shown in Figure 12.1b. In this graph — and in all graphs constructed in this way — every word will have exactly one incoming edge, except for the root word, which is indicated by a special incoming arrow from above. Another feature of this graph is that it is **weakly connected**, in the sense that if we replaced the directed edges with undirected edges, there would be a path between all pairs of nodes. From these properties, it can be shown that there are no cycles in the graph (or else at least one node would have to have more than one incoming edge), and therefore, the graph is a **tree**.

Although we have begun by motivating dependency grammar in terms of lexicalized constituent parsing, there is a rich literature on dependency grammar as a model of syntax in its own right (Tesnière, 1966). Kübler et al. (2009) provides a comprehensive overview of this literature.

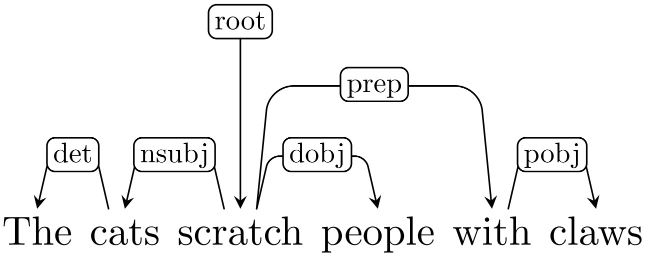


Figure 12.2: A labeled dependency parse

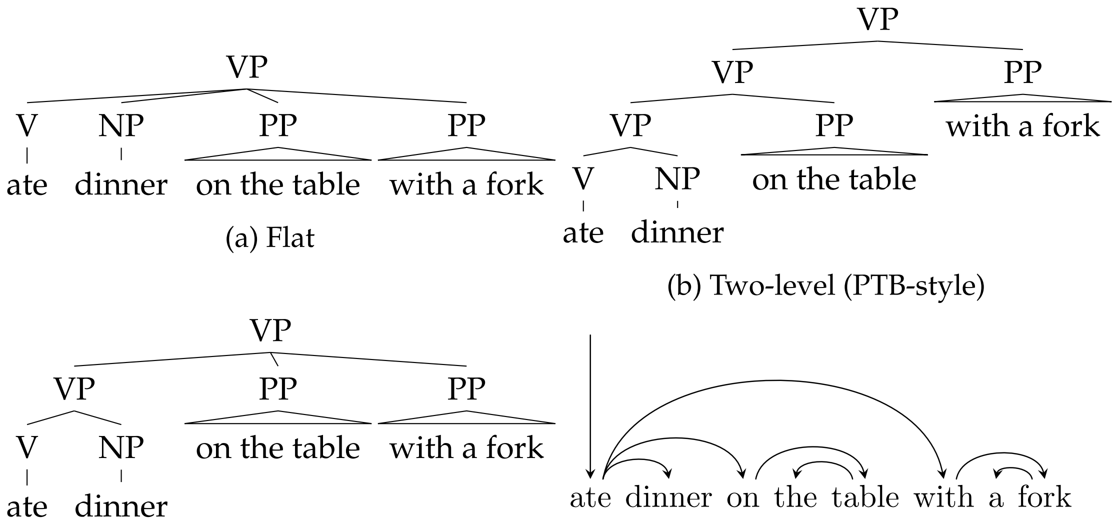
What do the edges mean?

A dependency edge implies an asymmetric syntactic relationship between the head and dependent words.For a pair like *the cats* or *cats scratch*, how do we decide which is the head? Here are some possible criteria:

- The head sets the syntactic category of the construction: for example, nouns are the heads of noun phrases, and verbs are the heads of verb phrases.
- The modifier may be optional while the head is mandatory: for example, in the sentence *cats scratch people with clause*, the substrings *cats scratch* and *cats scratch people* are grammatical sentences, but *with claws* is not.
- The head determines the morphological form of the modifier: for example, in languages that require gender agreement, the gender of the noun determines the gender of the adjectives and determiners.

As always, these guidelines sometimes conflict, but it is possible to use these basic principles define fairly consistent conventions at the level of part-of-speech tags, similar to the head percolation rules from lexicalized constituent parsing.

Edges may be **labeled** to indicate the nature of the syntactic relation that holds between the two elements. An example is shown in Figure 12.2. The edge between *scratch* and *cats* is labeled NSUBJ, with *scratch* as the head; this indicates that the noun subject of the predicate verb *scratch* is headed by the word *cats*. The edge from *scratch* to *people* is labeled with DOBJ; this indicates that the word *people* is the head of the direct object. The Stanford typed dependencies have become a standard inventory of dependency types for English (De Marneffe and Manning, 2008). De Marneffe et al. (2014) propose a more minimal "universal" set of dependencies that is suitable for many languages.



(a) Flat

(b) Two-level (PTB-style)

(c) Chomsky adjunction

(d) Dependency representation

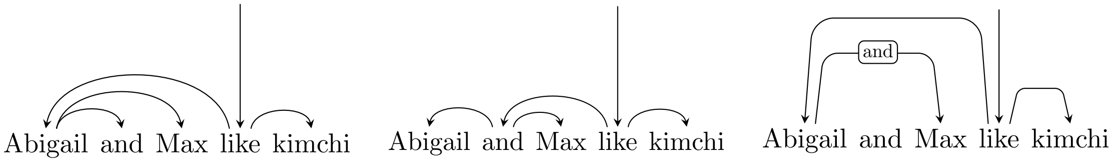
Figure 12.3: The three different CFG analyses of this verb phrase all correspond to a single dependency structure.

Ambiguity and difficult cases

The attachment ambiguity in the sentence shown in Figure 12.2 can be represented by a single change: replacing the edge from *scratch* to *with* by an edge from *people* to *with*. This should give you an idea of why labeled dependency trees are useful: they tell us who did what to whom.

However, dependency trees are less structurally expressive than lexicalized CFG derivations. That means they hide information that would be present in a CFG parse. Often this "information" is in fact irrelevant for any conceivable linguistic purpose: for example, Figure 12.3 shows three different ways of representing prepositional phrase adjuncts to the verb *ate*. Because there is apparently no meaningful difference between these analyses, the Penn Treebank decides by convention to use the two-level representation. As shown in Figure 12.3d, these three cases all look the same in a dependency parse. So if you didn't think there was any meaningful difference between these three constituent representations, you may view this as an advantage of the dependency representation.

Dependency grammar still leaves open some tricky representational decisions. For example, coordination is a challenge: in the sentence, *Abigail and Max like kimchi* (Figure 12.4), which word is the immediate dependent of the main verb *likes*?


Abigail and Max like kimchi
Abigail and Max like kimchi
and
Abigail and Max like kimchi

(a) The leftmost coordinated item is the head.

(b) The coordinating conjunction is the head.

(c) The coordinating conjunction is "collapsed" out.

Figure 12.4: Three alternatives for representing coordination in a dependency parse

Choosing either *Abigail* or *Max* seems arbitrary; for fairness we might choose *and*, but this seems in some ways to be the least important word in the noun phrase. One typical solution is to simply choose the left-most item in the coordinated structure — in this case, *Abigail*. Another alternative, as shown in Figure 12.4c, is a **collapsed** dependency grammar in which conjunctions are not included as nodes in the graph, but are instead used to label the edges (De Marneffe et al., 2006). Popel et al. (2013) survey alternatives for handling this phenomenon across several dependency treebanks.

The same logic that makes us reluctant to accept *and* as the head of a coordinated noun phrase may also make us reluctant to accept a preposition as the head of a prepositional phrase. In the sentence *cats scratch people with claws*, surely the word *claws* is more central than the word *with* — and it is precisely the bilexical relations between *scratch*, *claws*, and *people* that help guide us to the correct syntactic interpretation. Yet there are also arguments for preferring the preposition as the head — as we saw in section 11.5, the preposition itself is what helps us to choose verb attachment in *meet the President* ***on*** *Monday* and noun attachment in *meet the President* ***of*** *Mexico*. Collapsed dependency grammar is again a possible solution: we can collapse out the prepositions so that the dependency chain,

$$\textit{President} \rightarrow_{prep} \textit{of} \rightarrow_{pobj} \textit{Mexico}$$

would be replaced by *President* $\rightarrow_{PREP:of}$ *Mexico*.

Projectivity

The dependency graphs that can be built from all possible lexicalized constituent parses of a sentence with M words are a proper subset of the spanning trees over M nodes. In other words, there exist spanning trees that do not correspond to any

	% non-projective edges	% non-projective sentences
Czech	1.86%	22.42%
English	0.39%	7.63%
German	2.33%	28.19%

Table 12.1: Frequency of non-projective dependencies in three languages (Kuhlmann and Nivre, 2010)

lexicalized constituent parse. This is because syntactic constituents are **contiguous** spans of text, so that the head h of the constituent that spans the nodes from i to j must have a path to every node in this span. This property is known as **projectivity**. Informally, it means that "crossing edges" are prohibited. The formal definition follows:

Definition 2 (Projectivity). *An edge from i to j is projective iff all k between i and j are descendants of i. A dependency parse is projective iff all its edges are projective.*

If we were to annotate a dependency parse directly — rather than deriving it from a lexicalized constituent parse — such non-projective edges would occur. Figure 12.5 gives an example of a non-projective dependency graph in English. This dependency graph does not correspond to any constituent parse. In languages where non-projectivity is common, such as Czech and German, it is better to annotate dependency trees directly, rather than deriving them from constituent parses. An example is the Prague Dependency Treebank (Böhmová et al., 2003), which contains 1.5 million words of Czech, with approximately 12,000 non-projective edges (see Table 12.1). Even though relatively few dependencies are non-projective in Czech and German, many sentences have at least one such dependency.

As we will see in the next section, projectivity has important consequences for the sorts of algorithms that can perform dependency parsing.

12.2 Graph-based dependency parsing

Let $\boldsymbol{y} = \{\langle i, j, r\rangle\}$ indicate a dependency graph with relation r from head word w_i to dependent word w_j. We would like to define a scoring function $\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{y}, \boldsymbol{w})$, where $\boldsymbol{f}(\boldsymbol{y}, \boldsymbol{w})$ is a vector of features on the dependency graph and sentence, and $\boldsymbol{\theta}$ is a vector of weights. The dependency parsing problem is then the structure

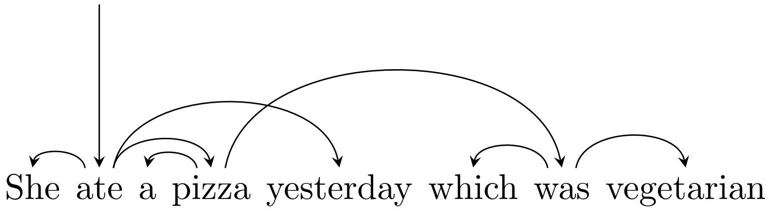


Figure 12.5: An example of a non-projective dependency parse in English

prediction problem,

$$\hat{\boldsymbol{y}} = \text{argmax}\, \boldsymbol{y} \in \mathcal{Y}(\boldsymbol{w}) \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{y}, \boldsymbol{w}). \tag{12.1}$$

As usual, the number of possible labelings $\mathcal{Y}(\boldsymbol{w})$ is exponential in the length of the input. In the case of non-projective dependency parsing, the set $\mathcal{Y}(\boldsymbol{w})$ includes all possible spanning trees over a complete graph with M nodes, where M is the length of the sentence $\boldsymbol{w}$. The size of this set is M^{M-2} (Wu and Chao, 2004). Algorithms that search over this space of possible graphs are known as **graph-based dependency parsers**.

In sequence labeling and constituent parsing, it was possible to search efficiently over an exponential space by choosing a feature function that decomposes into a sum of local feature vectors. A similar approach is possible for dependency parsing, by requiring the feature function to decompose across dependency arcs $i \to j$:

$$\boldsymbol{f}(\boldsymbol{y}, \boldsymbol{w}) = \sum_{\langle i,j,r \rangle \in \boldsymbol{y}} \boldsymbol{f}(\boldsymbol{w}, i, j, r) \tag{12.2}$$

$$\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{y}, \boldsymbol{w}) = \sum_{\langle i,j,r \rangle \in \boldsymbol{y}} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, i, j, r). \tag{12.3}$$

Dependency parsers that operate under this assumption are known as **arc-factored**, since the overall (exponentiated) score is a product of scores over all arcs. As described later in this section, the arc-factored assumption enables efficient algorithms for dependency parsing.

Features

Typical features for arc-factored dependency parsing are similar to those used in sequence labeling and discriminative constituent parsing. They include: the length and direction of the dependency arc; the words linked by the dependency relation; their prefixes, suffixes, and part-of-speech tags (as producted by an automatic tagger); and their neighbors in the sentence. In labeled dependency parsing, each of these features are also conjoined with the relation type r.

Bilexical features, which include both the head and the dependent, will be helpful for common words, but will be extremely sparse for rare words. It is therefore necessary to include features at various levels of detail, such as: word-word, word-tag, tag-word, and tag-tag. For example, for the arc *scratch* → *cats*, we might have the features,

$$\begin{aligned}\{w_i \to w_j : & \qquad \textit{scratch} \to \textit{cats}, \\ w_i \to t_j : & \qquad \textit{scratch} \to \text{NNS}, \\ t_i \to w_j : & \qquad \text{VBP} \to \textit{cats}, \\ t_i \to t_j : & \qquad \text{VBP} \to \text{NNS}\}\end{aligned}$$

Regularized discriminative learning algorithms can then learn to trade off between features that are rare but highly predictive, and features that are common but less informative.

As with sequence labeling, it is possible to includes features on neighboring words without breaking the locality restriction: we can consider features such as the identity, part-of-speech, and shape of the preceding and succeeding words, $w_{i-1}, w_{i+1}, w_{j-1}, w_{j+1}$. What we cannot do (yet) is consider other parts of the graph $\boldsymbol{y}$, such as the parent of i (which I will denote $w_{\Gamma(i)}$) or the siblings of j, the set $\{w_j : \Gamma(j) = i\}$. This requires higher-order dependency parsing, discussed in section 12.2.

To give a concrete example, the seminal paper by McDonald et al. (2005a) includes the following features for an arc between words w_i and w_j, with part-of-speech tags t_i and t_j:

Unigram features $\langle w_i \rangle; \langle t_i \rangle; \langle w_i, t_i \rangle; \langle w_j \rangle; \langle t_j \rangle; \langle w_j, t_j \rangle$.

Bigram features $\langle w_i, t_i, w_j, t_j \rangle; \langle w_i, w_j, t_j \rangle; \langle t_i, w_j, t_j \rangle; \langle w_i, t_i, t_j \rangle; \langle w_i, t_i, w_j \rangle; \langle w_i, w_j \rangle; \langle t_i, t_j \rangle$.

"In-between" features $\langle t_i, t_k, t_j \rangle$ for all k between i and j.

Neighbor features $\langle t_i, t_{i+1}, t_{j-1}, t_j \rangle; \langle t_{i-1}, t_i, t_{j-1}, t_j \rangle; \langle t_i, t_{i+1}, t_j, t_{j+1} \rangle; \langle t_{i-1}, t_i, t_j, t_{j+1} \rangle$

In addition, all the word features are supplemented with the five-character prefixes for all words longer than five characters (e.g., *unconscionable* → *uncon*). The bigram features include several varieties of backoff from the most detailed 4-tuple feature; McDonald et al. (2005a) note that these backoff features were particularly helpful, presumably because they improve generalization. The "in-between" features activate for all part-of-speech tags between positions i and j in the sentence. This feature group helps to "rule out situations when a noun would attach to another noun with a verb in between, which is a very uncommon phenomenon."

Learning

Having formulated graph-based dependency parsing as a structure prediction problem, we can apply similar learning algorithms to those used in sequence labeling. The most direct application is structured perceptron,

$$\hat{\boldsymbol{y}} = \operatorname{argmax} \boldsymbol{y}' \in \mathcal{Y}(\boldsymbol{w})\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}') \tag{12.4}$$

$$\boldsymbol{\theta}^\top = \boldsymbol{\theta}^\top + \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}) - \boldsymbol{f}(\boldsymbol{w}, \hat{\boldsymbol{y}}) \tag{12.5}$$

This is just like sequence labeling, but now $\operatorname{argmax} \boldsymbol{y}' \in \mathcal{Y}(\boldsymbol{w})$ requires a maximization over all dependency trees for the sentence. Algorithms for performing this search efficiently are described below. We can apply all the usual tricks from chapter 2: weight averaging, large-margin, and regularization. McDonald et al. (2005a,b) were the first to treat dependency parsing as a structure prediction problem, using MIRA (a close relative of the passive-aggressive algorithm we saw in chapter 2) to obtain high accuracy parses in both projective and non-projective settings.

Conditional random fields (CRFs) are globally-normalized conditional models (see chapter 6), and they can be applied to any graphical model in which we can efficiently compute marginal probabilities over individual random variables — in this case, we need marginals over the edges. The marginals are required because the unregularized log-likelihood has a gradient that sums over all possible edges, taking the difference between the features in the observed dependency parses and the expected feature counts under $\mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w})$:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}} = \sum_{(i,j) \in \mathcal{Y}} \boldsymbol{f}(\boldsymbol{w}, i, j) - \sum_{i,j} \mathrm{p}(i \to j \mid \boldsymbol{w}) \boldsymbol{f}(\boldsymbol{w}, i, j) \tag{12.6}$$

For projective dependency trees, the marginal probabilities can be computed in cubic time, using a variant of the inside-outside algorithm (Lari and Young, 1990). For non-projective dependency parsing, marginals can be computed in cubic time, using the matrix-tree theorem (Koo et al., 2007; McDonald et al., 2007; Smith and Smith, 2007). We will not explore algorithms for computing marginals in this chapter, but they are described in more detail by Kübler et al. (2009).

Algorithms for non-projective dependency parsing

In **non-projective dependency parsing**, the goal is to identify the highest-scoring spanning tree over the words in the sentence. The arc-factored assumption ensures that the score for each spanning tree will be computed as a sum over scores for the edges. We can precompute these scores, $\psi(i \to j, r) = \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, i, j, r)$, before applying a parsing algorithm. (We must compute $\mathcal{O}(M^2 R)$ such scores, where M is the length of the sentence and R is the number of dependency relation types, so this is a lower bound on the time complexity of any exact algorithm for dependency parsing.)

Algorithm 8 Chu-Liu-Edmonds algorithm for unlabeled dependency parsing

```
procedure CHU-LIU-EDMONDS($\{\psi(i \to j)\}_{i,j \in \{1\dots M\}}$)
    for $j \in 1 \dots M$ do
        $h_j \leftarrow \text{argmax}\, i\psi(i \to j)$
    $\tau \leftarrow \{j, h_j\}_{j \in 1\dots M}$
    $\mathcal{C} \leftarrow$ FINDCYCLES($\tau$)
    if $\mathcal{C} = \varnothing$ then return $\tau$
    else
        for each cycle $c \in \mathcal{C}$ do
            Remove all nodes in the cycle from the graph
            Add a “super-node” representing the cycle
        return CHU-LIU-EDMONDS($G$)[todo: how to show this?]
```

Based on these scores, we build a weighted connected graph. Arc-factored non-projective dependency parsing is then equivalent to finding the the spanning tree that achieves the maximum total score, $\sum_{\langle i,j,r\rangle \in \boldsymbol{y}} \psi(i \to j, r)$. The **Chu-Liu-Edmonds algorithm** (Chu and Liu, 1965; Edmonds, 1967) computes this spanning tree in time $\mathcal{O}(M^3 R)$. The algorithm, which is sketched in Algorithm 8, operates recursively. It first identifies the highest scoring incoming edge for each node, and then checks the graph for cycles. If there are no cycles, the resulting graph is a spanning tree, and moreover, it is the maximum spanning tree, because there is no better-scoring incoming edge for each node. If there is a cycle, the cycle is collapsed into a “super-node”, whose incoming edges have scores equal to the scores of the best spanning tree that includes both the edge and all nodes in the cycle.[todo: help].

The algorithm works because it can be proved that the maximum spanning tree on the contracted graph is equivalent to the maximum spanning tree on the

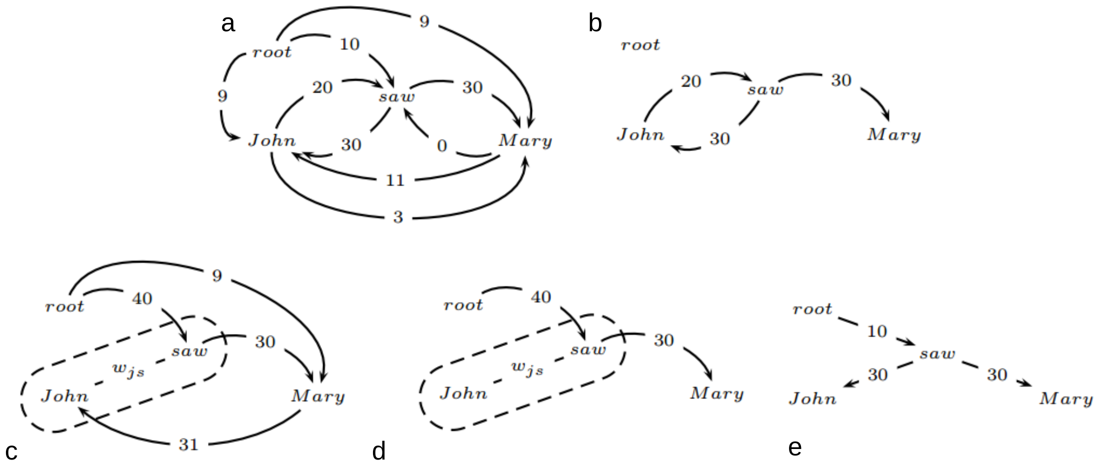


Figure 12.6: An illustration of the MST algorithm on a simple example. Figure borrowed from McDonald et al. (2005b).

original graph. The basic process is illustrated in Figure 12.6. In part (a), we see the complete graph, which includes all edge scores $\psi(i \to j)$. In (b), we see the highest scoring incoming edge for each node. In (c), the cycle between *John* and *saw* is contracted, creating new incoming edges with weight 40 from the root, and weight 31 from *Mary*. In (d), we find the highest-scoring incoming edge in the new graph. There are no remaining cycles, so we recover the maximum spanning tree.

Let us consider the time complexity of unlabeled dependency parsing first. For each of the M words in the sentence, one must search all $M-1$ other words for the highest-scoring incoming edge, for a time complexity of $\mathcal{O}(M^2)$. In the worst case, it is necessary to contract the graph M times. If we redo the search within each contraction, we face a total cost of $\mathcal{O}(M^3)$. Recall that the CKY constituent parsing algorithm is also cubic time complexity in the length of the sentence. However, further optimizations are possible, resulting in a complexity of $\mathcal{O}(M^2)$(Tarjan, 1977). To generalize the algorithm to labeled dependency parsing, it is necessary only to compute the best scoring label for each possible edge. Because of the arc-factoring assumption, the edge labels are decoupled from each other, so this can be done as a preprocessing step, with total complexity of $\mathcal{O}(M^2 R)$.

Algorithms for projective dependency parsing

The Chu-Liu-Edmonds algorithm finds the best scoring dependency tree, but it does not enforce the projectivity constraint. For languages in which we expect projectivity — such as English — we may prefer to ensure that the parsing algorithm returns only projective trees. Note that the arc-factored assumption makes it impossible to **learn** to produce projective trees, since projectivity cannot be encoded in a feature that decomposes over individual arcs.

Recall that it is possible to convert any lexicalized constituent parse directly into a projective dependency parse. This means that any algorithm for lexicalized constituent parsing is also an algorithm for projective dependency parsing. One such algorithm is presented in section 11.5, in which we built a table where the cell $t[i, j, h, X]$ contains the score of the best derivation of the substring $\boldsymbol{w}_{i:j}$ from non-terminal X, in which the head is w_h. For unlabeled projective dependency parsing, we can apply a very similar algorithm:

$$\begin{aligned}
t_\ell[i, j, h] =& \max_{k > h} \max_{k \leq h' < j} t[i, k, h] + t[k, j, h'] + \psi(h \rightarrow h') && (12.7)\\
t_r[i, j, h] =& \max_{k \leq h} \max_{i \leq h' < k} t[i, k, h'] + t[k, j, h] + \psi(h \rightarrow h') && (12.8)\\
t[i, j, h] =& \max\left(t_\ell[i, j, h], t_r[i, j, h]\right). && (12.9)
\end{aligned}$$

The goal is for $t[i, j, h]$ to contain the score of the best-scoring projective dependency tree for $\boldsymbol{w}_{i:j}$, headed by w_h. We must first maximize over all h', which is the location of an immediate dependent of w_h. Projectivity guarantees that the subtree headed by h' will extend to one of the endpoints of the entire span: either from the left endpoint i to some midpoint k, or from some midpoint k to the right endpoint j. We compute the best score for each of these possibilities separately in Equation 12.7 and Equation 12.8. Computing each of these scores also involves maximizing over all possible midpoints k.

We construct the table t from the bottom up: first compute scores for all subtrees of size 2, then size 3, and so on. The total size of the table is $\mathcal{O}(M^3)$, and to complete each cell we must search over $\mathcal{O}(M)$ dependents and $\mathcal{O}(M)$ split points.

Thus, the overall complexity if $\mathcal{O}(M^5)$. The Eisner (1996) algorithm reduces this complexity to $\mathcal{O}(M^3)$ by maintaining multiple tables. For a detailed description of this algorithm, see Kübler et al. (2009). As with the Chu-Liu-Edmonds algorithm, the best-scoring label for each edge can be computed as a preprocessing step, with complexity $\mathcal{O}(M^2R)$.

Higher-order dependency parsing

Arc-factored dependency parsers can only score dependency graphs as a product across their edges. However, it can be useful to consider higher-order features, which consider pairs or triples of edges, as shown in Figure 12.7. Second-order features consider **siblings** and **grandchildren**; third-order features consider **grand-siblings** (siblings and grandparents together) and **tri-siblings**.

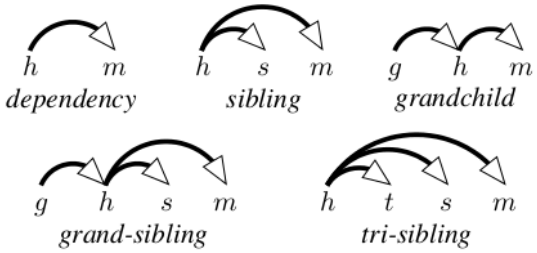


Figure 12.7: Feature templates for higher-order dependency parsing (Koo and Collins, 2010)

Why might we need higher-order dependency features? Consider the example *cats scratch people with claws*, where the preposition *with* could attach to either *scratch* or *people*. In a lexicalized first-order arc-factored dependency parser, we would have the following feature sets for the two possible parses:

- $\langle$*ROOT* $\rightarrow$ *scratch*$\rangle$, $\langle$*scratch* $\rightarrow$ *cats*$\rangle$, $\langle$*scratch* $\rightarrow$ *people*$\rangle$, $\langle$*scratch* $\rightarrow$ *with*$\rangle$, $\langle$*with* $\rightarrow$ *claws*$\rangle$
- $\langle$*ROOT* $\rightarrow$ *scratch*$\rangle$, $\langle$*scratch* $\rightarrow$ *cats*$\rangle$, $\langle$*scratch* $\rightarrow$ *people*$\rangle$, $\langle$*people* $\rightarrow$ *with*$\rangle$, $\langle$*with* $\rightarrow$ *claws*$\rangle$

The only difference between the feature vectors are the features $\langle$*scratch* $\rightarrow$ *with*$\rangle$ and $\langle$*people* $\rightarrow$ *with*$\rangle$, but both are reasonable features, both syntactically and semantically. A first-order arc-factored dependency parsing model would therefore struggle to find the right solution to this sentence. However, if we add grandcihld features, then our feature sets include:

- $\langle$*scratch* → *with* → *claws*$\rangle$
- $\langle$*people* → *with* → *claws*$\rangle$,

The first feature is preferable, so a second-order dependency parser would have a better chance of correctly parsing this sentence. In general, higher-order features can yield substantial improvements in dependency parsing accuracy (e.g., Koo and Collins, 2010).

Projective second-order parsing can still be performed in $\mathcal{O}(M^3)$ time (and $\mathcal{O}(M^2)$ space), using a modified version of the Eisner algorithm. Projective third-order parsing can be performed in $\mathcal{O}(M^4)$ time and $\mathcal{O}(M^3)$ space (Koo and Collins, 2010). Approximate pruning algorithms can reduce this cost significantly by filtering out unpromising edges (Rush and Petrov, 2012).

Given the tractability of higher-order projective dependency parsing, you may be surprised to learn that non-projective second-order dependency parsing is NP-Hard! This can be proved by reduction from the vertex cover problem (Neuhaus and Bröker, 1997). One heuristic solution is to do projective parsing first, and then post-process the projective dependency parse to add non-projective edges (Nivre and Nilsson, 2005). More recent work has applied advanced techniques for approximate inference in graphical models, including belief propagation (Smith and Eisner, 2008), integer linear programming (Martins et al., 2009), variational inference (Martins et al., 2010), and Markov Chain Monte Carlo (Zhang et al., 2014).

12.3 Transition-based dependency parsing

Graph-based dependency parsing offers exact inference, meaning that it is possible to recover the best-scoring parse. But this exactness comes at a price: we can use only a limited set of features. These limitations are felt more keenly in dependency parsing than in sequence labeling; as we have already seen, second-order dependency features are critical to correctly identify certain types of attachments. We may also criticize graph-based parsing on the basis of intuitions about human language processing: people read and listen to sentences *sequentially*, incrementally building mental models of the sentence structure and meaning before getting to the end (Jurafsky, 1996). This seems hard to reconcile with graph-based algorithms, which perform bottom-up operations on the entire sentence, seemingly requiring the parser to keep every word in memory.

Transition-based algorithms address both of these objections. They work by moving through the sentence sequentially, while incrementally updating a stored representation of what has been read thus far. After processing the entire sentence, they return an analysis of its syntactic structure.

A simple transition-based parser is **shift-reduce**, an algorithm that you may have seen ...

An alternative to exact global inference is transition-based parsing: making a series of local decisions. We can apply a shift-reduce algorithm, just as we considered for CFG parsing in **??**. The reduce actions are different: rather than combining elements into non-terminals, they create arcs between words, leaving the head of edge.

- **shift**: push a word onto the stack
- **right-reduce**: make a right-facing edge between the top two elements on the stack
- **left-reduce**: make a left-facing edge between the top two elements on the stack
- Alternatively, "arc-eager" dependency parsing distinguishes **reduce** from **arc-right** and **arc-left**, which create arcs between the top of the stack and the first element in the queue. Arc-eager parsing is arguably more cognitively plausible, because it constructs larger connected components incrementally, rather than having a deep stack with lots of disconnected elements (Abney and Johnson, 1991; Nivre, 2004).

Shift-reduce potentially suffers from search errors, since an early mistake can make it impossible to find the best-scoring parse. However, it has been shown to be both accurate and fast (Nivre, 2004; Nivre et al., 2007) — the time complexity is linear in the length of the sentence! Another advantage of shift-reduce is that there is no restriction on the features that can be considered to make each parsing decision.

Beam search is an improvement on shift-reduce, with the goal of eliminating search errors. As we move through the sentence, we keep a **beam** of possible hypotheses; at each stage, we keep the k best unique hypotheses on the beam.

Learning transition based dependency parsers

[todo: needs way more detail] For transition-based dependency parsing, learning means training a classifier to make the correct shift and reduce decisions. We

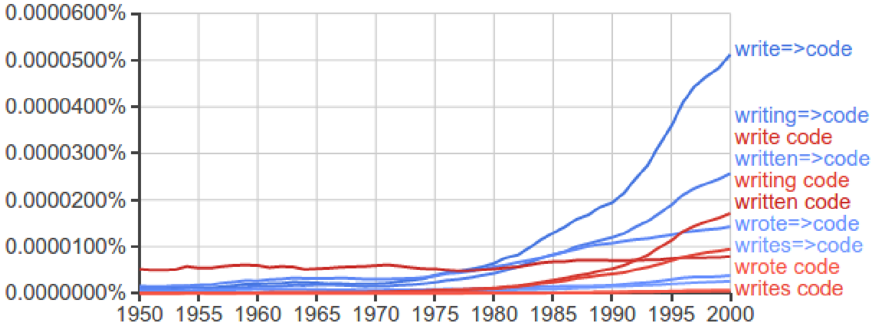


Figure 12.8: Google n-grams results for the bigram *write code* and the dependency arc *write* => *code* (and their morphological variants)

can do this by identifying a series of decisions that is required to produce the correct dependency parse.[1] We can build a training set by treating each decision in the derivation of the correct parse is a positive instance, and every other possible decision is a negative instance. However, Huang et al. (2012) offer alternative perceptron learning rules that yield improvements when learning in the beam search setting.

A key advantage of transition-based parsing is that there is no restriction to arc-factored features; we can include any feature of the current partial parse, history of decisions, etc. It is also fast: linear time in the length of the sentence.

12.4 Applications

Dependency parsing is used in many real-world applications: any time you want to know about pairs of words which might not be adjacent, you can use dependency links instead of typical regular expression search patterns. For example, we may want to match strings like *delicious pastries*, *delicious French pastries*, and *the pastries are delicious*[2]

It is now possible to search Google n-grams by dependency edges; for example, finding the trend in how often a dependency edge has appeared over time. For example, we might be interested in knowing when people started talking

[1] **Spurious ambiguity** occurs when multiple decision sequences give the same dependency parse.

[2] Note that the copula *is* is collapsed in many dependency parsing systems, such as the Stanford dependency parser De Marneffe and Manning (2008).

about *writing code*, but we also want *write some code*, *write the code*, *write all the code*, etc. By searching on dependency edges, we can recover this information, as shown in Figure 12.8. This capability has implications for research in digital humanities, as shown by the analysis of Shakespeare performed by Muralidharan and Hearst (2013).

A classic application of dependency parsing is **relation extraction**, which is described in chapter 18. The goal of relation extraction is to identify entity pairs, such as

$$\langle \text{Tolstoy}, \text{War and Peace} \rangle$$
$$\langle \text{Marquéz}, \text{100 Years of Solitude} \rangle$$
$$\langle \text{Shakespeare}, \text{A Midsummer Night's Dream} \rangle,$$

which stand in some relation to each other (in this case, the relation is authorship). Such entity pairs are often referenced via consistent chains of dependency relations. Therefore, dependency paths are often a useful feature in supervised systems which learn to detect new instances of a relation, based on labeled examples of other instances of the same relation type (Culotta and Sorensen, 2004; Fundel et al., 2007; Mintz et al., 2009).

Cui et al. (2005) show how dependency parsing can improve question answering. For example, you might ask,

(12.1) *What % of the nation's cheese does Wisconsin produce?*

Now suppose your corpus contains this sentence:

(12.2) *In Wisconsin, where farmers produce 28% of the nation's cheese, . . .*

The location of *Wisconsin* in the surface form of this string might make it a poor match for the query. However, in the dependency graph, there is an edge from *produce* to *Wisconsin* in both the question and the potential answer, raising the likelihood that this span of text is relevant to the question.

A final example comes from sentiment analysis. As discussed in chapter 3, the polarity of a sentence can be reversed by negation, e.g.

(12.3) *There is no reason at all to believe the polluters will suddenly become reasonable.*

By tracking the sentiment polarity through the dependency parse, we can better identify the overall polarity of the sentence, determining when key sentiment words are reversed (Wilson et al., 2005; Nakagawa et al., 2010).

Part III

Meaning

Chapter 13

Logical semantics

A grand ambition of natural language processing, and indeed, all of artificial intelligence, is to convert natural language into a representation that supports **semantic inferences**.[1] Many applications of language technology involve some level of semantic understanding:

- Answering questions. This includes "real-life" questions like *where can a guy find a decent cup of coffee around here?*, and also "quiz show" questions like *what's the middle name of the mother of the 44th President of the United States?*
- Translating a sentence from one language into another, while preserving the underlying meaning.
- Building a robot that can follow natural language instructions and execute useful tasks.
- Fact-checking an article by searching the web for contradictory evidence.
- Logic-checking an argument by trying to identify contradictions or unsupported assertions.

Most approaches towards achieving this level of semantic understanding involve converting natural language to some form of **meaning representation**. Jurafsky and Martin (2009) compare several alternative representations, showing parallels between several representations that are superficially distinct. Therefore, we will focus on logical representations: **boolean logic**, **first-order logic**, and the **lambda calculus**.

[1] Alternative readings on this topic include the chapter from Jurafsky and Martin (2009), a more involved "informal" reading from Levy and Manning (2009), and a yet more involved introduction from Briscoe (2011).

13.1 Meaning representations

The goal of a meaning representation is to provide a way to express **propositions**, while abstracting over the ambiguity and vagueness of natural language. There are several criteria that a meaning representation should meet:

Verifiability It should be possible to test the truth of assertions in the meaning representation. Indeed, in **truth-conditional semantics**, the meaning of a sentence is said to be identical to its truth conditions: that is, to the set of facts that must hold in the world for the sentence to be true.

We might imagine that verifiability should be tested against the real world: for example, if faced with the proposition *Alice hates apples*, we could verify it by finding Alice and asking her. However, it is better still to be able to reason about **possible worlds**, such as fictional worlds in which *Alice* (or *apples*) might refer to arbitrarily different entities. In **model-theoretic semantics**, each proposition has a **denotation** in a model of the world, enabling propositions to be verified against specific models corresponding to possible worlds. Why is this useful? Consider that Lois Lane is unaware that Superman and Clark Kent are the same person — that is, SUPERMAN and CLARKKENT have different denotations in her model. Model-theoretic semantics makes it possible to interpret statements from her perspective, so that, for example, it would not be absurd for her to ask Clark to speak with Superman.[2]

Truth-conditional semantics allows us to define additional concepts of **equivalence** and **entailment**. A statement P is entailed by statement Q iff the truth conditions for Q imply the truth conditions for P. For example, the statement *Alice gives Bob a book about calculus* entails the statements *Alice gives Bob a book*, *Alice gives someone a book*, *Someone gives Bob a book*, etc. Iff P entails Q and Q entails P, then we can say that P and Q are logically equivalent.

No ambiguity Each sentence in the meaning representation should have exactly one meaning. In truth conditional semantics, this means that each sentence in the meaning representation has exactly one corresponding set of truth conditions.

Clearly this criterion is not met by natural language. Many of the syntactic ambiguities that we encountered in previous sections have corresponding

[2] Example from Percy Liang's slides on semantics, http://icml.cc/2015/tutorials/icml2015-nlu-tutorial.pdf

semantic ambiguities: consider the truth conditions for the two possible PP attachments in our example *cats scratch people with claws*, or the example *she fed her dog biscuits*. Natural language also has ambiguity at the lexical level: the sentence *Dong bought a plant* would have distinct truth conditions depending on whether *plant* refers to something like a shrub, or a factory for producing widgets.

Jurafsky and Martin (2009) mention a converse criterion, **canonical form**, which requires that each meaning (set of truth conditions) has a single representation. For example, if we consider the database query language SQL as a meaning representation, then it is easy to design superficially distinct queries that will return the same results regardless of what database they are applied to:

(13.1)
```
SELECT RestaurantID, City FROM Restaurants WHERE City
= 'Atlanta' OR City = 'New York'
```

(13.2)
```
SELECT RestaurantID, City FROM Restaurants WHERE City
= 'New York' OR City = 'Atlanta'
```

In general, it is difficult to design meaning representations in which every meaning has a single canonical form. However, removing unnecessary flexibility can vastly simplify the computation associated with verifying statements and performing **inference** (described below).

Expressiveness Meaning representation is useful only to the extent that it enables us to talk about a wide range of different things. This is partly a matter of the **non-logical vocabulary** that the representation includes: the set of entities (e.g., *Alice*, *Bob*) and relations (e.g., *likes*, *brother-of*) that can be included in sentences. However, there are also deeper structural limits on expressiveness. Consider the following possibilities:

(13.3) *Alice admires Bob*

(13.4) *Alice admires Bob and Bob trusts Alice*

(13.5) *Alice admires someone*

(13.6) *Alice admires someone who trusts her*

(13.7) *Everyone whom Alice admires trusts someone*

(13.8) *Not everyone whom Alice admires trusts Bob*

To handle all of these cases, we must have an appropriate **logical vocabulary**, including boolean connectives and quantifiers. More on this in section 13.2.

Inference We would like to be able to combine assertions in our meaning representation to infer new facts about the world. For example, given the assertion *Bart is Lisa's brother*, we should be able to infer that *Someone is Lisa's brother*. Given the additional information that Lisa is female, we should be able to infer that *Lisa is Bart's sister* — although this inference is of a different type, since it requires additional knowledge about the relations BROTHER and SISTER.

How do natural languages like English do on these criteria? They are infinitely expressive, but highly ambiguous. Because we cannot establish the truth conditions of natural language expressions without ambiguity, it is difficult to speak of verifying their meaning or drawing further inferences.

But if natural language is not itself a meaning representation, we would still like to be able to find the most likely meaning, or the set of possible meanings, for a given natural language sentence. This task is known as **semantic parsing**, and it typically rests on the assumption that meaning is determined **compositionally**, with the meaning of a sentence determined by the meanings of its constituent expressions, and the operations that are used to combine them. In particular, we will assume that the relevant substrings of a sentence correspond to the syntactic constituents identified during CFG-style parsing, and that each parsing production corresponds to some semantic operation. More on this in section 13.3.

13.2 Logical representations of meaning

We will build a meaning representation on logical semantics, which does a pretty good job of meeting the criteria established in the previous section.

Propositional logic

The bare bones of logical meaning representation are boolean operations on propositions:

Propositional symbols We use the symbols $P, Q, \ldots$ to represent propositions; for example, P may correspond to the proposition, *bagels are delicious*.

Boolean operators We can evaluate the truth of more complex statements through boolean operators: negation ($\neg P$, which is true if P is false), conjunction ($P \wedge Q$, which is true if both P and Q are true), and disjunction ($P \vee Q$, which is true if at least one of P and Q is true). Other operators can be derived from these: for example, implication ($P \Rightarrow Q$) has identical truth conditions to $\neg P \vee Q$; equivalence ($P \Leftrightarrow Q$) has identical truth conditions to $(P \wedge Q) \vee (\neg P \wedge \neg Q)$. In fact, if we have $\neg$, then only one of $\wedge$ and $\vee$ is needed; we can derive the other.

We can define axioms or inference rules in terms of these boolean connectives (communtativity, associativity, etc), and then derive further equivalences, which can support some inferences. For example, suppose $P =$ *The music is loud* and $Q =$ *Max can't sleep*. Then if we have $P \Rightarrow Q$ (*If the music is loud, Max can't sleep*) and P (*the music is loud*), then we have Q (*Max can't sleep*). However, there are other inferences that we cannot perform with propositional logic alone. For example, let $R =$ *The music is quiet*; then we might hope that $R \Rightarrow \neg P$, but this is not supported without knowing more about the propositions themselves. For this, we turn to predicate logic.

Predicate logic

Predicate logic extends our meaning representation with several additional classes of terms:

Constants These are elements that name individual entities in the model, such as MAX and THEMUSIC. We say that the **denotation** of each constant in a model $\mathcal{M}$ is an element in the model, e.g., $[\![\text{MAX}]\!] = \texttt{d}$ and $[\![\text{THEMUSIC}]\!] = \texttt{m}$.

Predicates Predicates can be thought of as sets of objects, or equivalently, as functions from objects to truth values. For example CANSLEEP is a predicate, and we may have $[\![\text{CANSLEEP}]\!] = \{\texttt{d}, \texttt{e}, \ldots\}$, denoting the set of individuals who can sleep. We can then test the proposition CANSLEEP(MAX) by asking whether $[\![\text{MAX}]\!] \in [\![\text{CANSLEEP}]\!]$.

Functions Functions can be thought of as sets of pairs of objects, or equivalently, as functions from one object to another. For example BROTHER-OF is a function, so that $[\![\text{BROTHER-OF(LISA)}]\!] = [\![\text{BART}]\!]$.

We can now express statements like

$$\text{ISQUIET(THEMUSIC)} \Leftrightarrow \neg\text{ISLOUD(THEMUSIC)}, \tag{13.1}$$

but this only applies to a specific constant, THEMUSIC, and not more generally. For example, we might prefer to say that *anything* that is quiet is not loud. To make such general statements, we will need two additional elements in our meaning representation:

Variables These are mechanisms for referring to objectives, which are not locally specified. We can then write BROTHER-OF(x) or ISLOUD(x), using x here as an **unbound variable**.

Quantifiers To bind variables, we use quantifiers. Variables can be used to refer to some particular unspecified object, or to all possible objectives. Correspondingly, we have two connectives, $\exists$ and $\forall$. The statement,

$$\exists x : \text{BROTHER-OF(LISA)} = x, \tag{13.2}$$

uses the **existential quantifier** $\exists$ to assert that there is at least one object which is the brother of Lisa in the model. The statement,

$$\forall x : \text{ISLOUD}(x) \Leftrightarrow \neg\text{ISQUIET}(x) \tag{13.3}$$

uses the **universal quantifier** $\forall$ to generalize the relationship between the predicates ISLOUD and ISQUIET; for this sentence to be true, it must be the case that for all entities in the model, the predicate ISLOUD only holds in exactly those cases in which the predicate ISQUIET does not hold.

Lambda calculus

Predicate logic is verifiable, unambiguous, expressive enough for a wide range of statements, and supports inferences; it does a good job meeting all of the criteria listed at the beginning of the chapter. But we still need a few more pieces before we can build logical meanings from natural language sentences.

Recall the assumption of **compositionality**, which states that the meaning of a natural language sentence is composed from the meaning of its constituents. Now, a simple sentence like *Max likes dragons* has two top-level constituents in a CFG parse: the NP *Max*, and the VP *likes dragons*. The meaning of *Max* is the constant MAX, and the meaning of the entire sentence might be LIKES(MAX,DRAGONS). But what is the meaning of the VP constituent *likes dragons*?

We will think of the meaning of VPs such as *likes dragons* as **functions** which require additional arguments to form a sentence in predicate logic. The notation for describing such functions is called **lambda calculus**, and it involves expressions such as $\lambda x.P(x)$, which indicates a function that takes an argument x and then has value $P(x)$. The application of a function $\lambda x.P(x)$ to an argument A is written

$$\lambda x.P(\ldots, x, \ldots)(A) \tag{13.4}$$

$$P(\ldots, A, \ldots), \tag{13.5}$$

indicating that A is playing the role occupied by the variable x, which is bound here by the lambda expression. It is crucial to note that P itself may be a lambda expression, so that application can be performed multiple times.

13.3 Syntax and semantics

We will now extent CFG products to include the meaning of each constituent, using rules of the form,

$$X : \alpha \rightarrow Y : \beta \quad Z : \gamma, \tag{13.6}$$

where X, Y, Z are syntactic non-terminals and α, β, γ are the meanings associated with each constituent.

For example, consider the very simple fragment,

$$\text{S} : \beta(\alpha) \rightarrow \text{NP} : \alpha \quad \text{VP} : \beta \tag{13.7}$$

$$\text{VP} : \beta(\alpha) \rightarrow \text{V} : \beta \quad \text{NP} : \alpha \tag{13.8}$$

$$\textit{Abigail}, \text{NP} \; : \textsc{Abigail} \tag{13.9}$$

$$\textit{Max}, \text{NP} \; : \textsc{Max} \tag{13.10}$$

$$\textit{likes}, \text{V} \; : \lambda y.\lambda x.\textsc{like}(x, y) \tag{13.11}$$

Lines 13.9-13.11 describe the **lexicon**, listing the syntactic categories and semantic meanings of individual words. Words may have multiple entries in the lexicon, depending on their semantics; for example, the verb *eats* may be intransitive (*Abigail eats*) or transitive (*Abigail eats kimchi*), so we need two lexical entries:

$$\textit{eats}, \text{V} \; : \lambda x.\textsc{eat}(x) \tag{13.12}$$

$$\textit{eats}, \text{V} \; : \lambda y.\lambda x.\textsc{eat}(x, y). \tag{13.13}$$

Now, given the sentence *Max likes Abigail*, we get the following analysis,

$$P = \lambda y.\lambda x.\textsc{likes}(x, y)(\textsc{Max})(\textsc{Abigail}) \tag{13.14}$$

$$= \lambda x.\textsc{likes}(x, \textsc{Abigail})(\textsc{Max}) \tag{13.15}$$

$$= \textsc{likes}(\textsc{Max}, \textsc{Abigail}) \tag{13.16}$$

Noun phrases

What about sentences with more complex noun phrases like *Max has a red bear* or *Abigail eats all the spicy snacks*? To handle these cases, we'll need to deal with determiners, adjectives, and general nouns. Let's start with a relatively simple case,

(13.9) *A dog likes Max.*

The desired analysis is,

$$(\textit{A dog likes Max.}).sem = \exists x.\textsc{dog}(x) \wedge \textsc{likes}(x, \textsc{Max}), \tag{13.17}$$

where $(\textit{text}).sem$ indicates the semantics of *text*.

We already know that the meaning of the verb phrase *likes Max* is $\lambda x.\textsc{likes}(x, \textsc{Max})$, and we would like to apply this function to the argument specified by the noun phrase. But somehow we have to get to a solution where the outermost term is the existential quantifier $\exists x$, and not the predicate LIKES. How can we do it?

The solution is to introduce some additional operations for **type-shifting**. The semantic type of the verb phrase *likes Max* was a function mapping from entities to truth values, $\lambda x.\textsc{likes}(x, \textsc{Max})$. We now introduce the type-**raising** operation $\alpha \rightarrow \lambda P.P(\alpha)$, indicating that the semantics α can be replaced with a function that takes P as an argument, and returns $P(\alpha)$. Applying type-raising to the verb phrase *likes Max*, we obtain, $\lambda P.P(\lambda x.\textsc{likes}(x, \textsc{Max}))$.

Now, how should we think of the noun phrase *a dog*? The determiner implies an existential quantifier (there exists some dog...) over all dogs, $\exists x.\textsc{dog}(x)$. Moreover, we are planning to apply some additional functions to explain what this dog is doing. So the semantics we want is $\lambda P.\exists(x)\textsc{dog}(x) \wedge P(x)$. We can get there by appropriately defining the determiner *a*, and the production NP → DET NN.

$$\text{NP} : \beta(\alpha) \rightarrow \textsc{Det} : \beta \quad \textsc{Nn} : \alpha \tag{13.18}$$

$$\begin{aligned} \textit{a}, \textsc{Det} &: \lambda P.\lambda Q.\exists x.P(x) \wedge Q(x) && (13.19)\\ \textit{dog}, \textsc{Nn} &: \lambda x.\textsc{dog}(x) && (13.20)\end{aligned}$$

Note that although we have typically treated the noun as the head of a noun phrase, it is the determiner whose semantics takes precedence in Equation 13.18. This enables us to properly assess the meaning of the phrase *a dog*,

$$\begin{aligned} (\textit{a dog}).sem =& (\lambda P.\lambda Q.\exists x.P(x) \wedge Q(x))(\lambda x.\textsc{dog}(x)) && (13.21)\\ =& \lambda Q.\exists(x).\textsc{dog}(x) \wedge Q(x) && (13.22)\end{aligned}$$

So now we have the two pieces,

$$\begin{aligned}
(\textit{a dog}).sem =&\lambda Q.\exists(x).\textsc{dog}(x) \wedge Q(x) && (13.23)\\
(\textit{likes Max}).sem =&\lambda x.\textsc{likes}(x, \textsc{Max}) && (13.24)\\
=&\lambda P.P(\lambda x.\textsc{likes}(x, \textsc{Max})), && (13.25)
\end{aligned}$$

using type-raising on the verb phrase. We can now combine the pieces, using the verb phrase semantics as a function on the noun phrase,

$$\begin{aligned}
(\textit{a dog likes Max}).sem =&(\lambda P.P(\lambda x.\textsc{likes}(x, \textsc{Max})))(\lambda Q.\exists(x).\textsc{dog}(x) \wedge Q(x)) && (13.26)\\
=&(\lambda Q.\exists(x).\textsc{dog}(x) \wedge Q(x))(\lambda x.\textsc{likes}(x, \textsc{Max})) && (13.27)\\
=&\exists(x).\textsc{dog}(x) \wedge \textsc{likes}(x, \textsc{Max}), && (13.28)
\end{aligned}$$

which is the desired semantics that we identified above for this sentence. A useful exercise is to try to do the same kind of analysis for the sentence *Max likes a dog*.

$$\begin{aligned}
(\textit{a dog}).sem =&\lambda P.\exists x.P(x) \wedge \textsc{dog}(x) && (13.29)\\
(\textit{likes}).sem =&\lambda y.\lambda z.\textsc{likes}(z, y) && (13.30)\\
=&\lambda Q.Q(\lambda y.\lambda z.\textsc{likes}(z, y)) && (13.31)\\
(\textit{likes a dog}).sem =&(\lambda Q.Q(\lambda y.\lambda z.\textsc{likes}(z, y)))(\lambda P.\exists x.P(x) \wedge \textsc{dog}(x)) && (13.32)\\
=&(\lambda P.\exists x.P(x) \wedge \textsc{dog}(x))(\lambda y.\lambda z.\textsc{likes}(z, y)) && (13.33)\\
=&\exists x.(\lambda y.\lambda z.\textsc{likes}(z, y))(x) \wedge \textsc{dog}(x) && (13.34)\\
=&\exists x.\lambda z.\textsc{likes}(z, x) \wedge \textsc{dog}(x) && (13.35)\\
(\textit{Max likes a dog}).sem =&\exists x.\textsc{likes}(\textsc{Max}, x) \wedge \textsc{dog}(x) && (13.36)
\end{aligned}$$

[todo: double-check this]

Full semantic analysis of natural language requires handling many more phenomena, but the basic strategy of function application and type-shifting covers much of what is needed. Jurafsky and Martin (2009) provide more details than presented here, and a book-length treatment is offered by Blackburn and Bos (2005).

13.4 Semantic parsing

The goal of **semantic parsing** is to convert natural language statements to a representation such as predicate logic with lambda calculus. Zettlemoyer and Collins

(2005) show that it is possible to train such a system, using labeled data of natural language sentences and their associated logical meanings. They use a linear model, in which each syntactic-semantic production has an associated feature weight, which is learned from labeled data.[3] A key point is that a sentence may be have analyses that produce the same logical interpretation, which is known as **spurious ambiguity**. They do not have labeled data for the specific productions, so they treat this is as a latent variable, and learn using a latent variable perceptron, where

$$\boldsymbol{z}^* = \underset{\boldsymbol{z}}{\operatorname{argmax}}\, \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}, \boldsymbol{z}) \tag{13.37}$$

$$\hat{\boldsymbol{y}}, \hat{\boldsymbol{z}} = \underset{\boldsymbol{y},\boldsymbol{z}}{\operatorname{argmax}}\, \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}, \boldsymbol{z}) \tag{13.38}$$

$$\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} + \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}, \boldsymbol{z}^*) - \boldsymbol{f}(\boldsymbol{w}, \hat{\boldsymbol{y}}, \hat{\boldsymbol{z}}), \tag{13.39}$$

with $\boldsymbol{y}$ indicating the logical interpretation and $\boldsymbol{z}$ indicating the derivation of that interpretation from the input $\boldsymbol{w}$.

A more ambitious approach is to train a semantic parser not from sentences annotated by their logical forms, but rather, from question-answer pairs, e.g., $\langle$*Where is Georgia Tech?*, *Atlanta*$\rangle$. There are now two latent variables: the logical form y, and the derivation of that logical form, z. We constrain the logical form y such that its denotation $[\![y]\!]$ is identical to the denotation of the logical form of the answer, e.g.,

$$[\![\lambda x.\textsc{Located-in}(\textsc{GeorgiaTech}, x)]\!] = [\![\textsc{Atlanta}]\!]. \tag{13.40}$$

This idea has been implemented by Clarke et al. (2010) and Liang et al. (2013), yielding systems that can answer questions about geographical relationships with above 90% accuracy.

[3] Zettlemoyer and Collins (2005) do not use context-free grammar, but instead use a mildly context-sensitive formalism called **Combinatory Categorial Grammar** (CCG). Semantic parsing is considerably easier to explain in CCG, but would require introducing a new syntactic formalism.

Chapter 14

Shallow semantics

“Full” compositional semantics requires representations at least as expressive as first-order logic. Machine learning approaches have improved robustness, and recent work has driven down the requirements for manually-created resources. But coverage is still relatively limited, with best performance in narrow domains like travel and geography.

Shallow semantics comprises a set of alternative approaches, which trade the expressiveness of representations like first-order logic for shallower representations which can be parsed more robustly, with broader coverage.

14.1 Predicates and arguments[1]

Shallow semantics focuses on predicate-argument relations. For example, the sentence *Abigail trusts Max* can be interpreted as trusts(ABIGAIL, MAX), where trusts is a predicate and ABIGAIL and MAX are its arguments. This is exactly the sort of relation that we saw in first-order logical semantics too, but in shallow semantics we will typically work without variables and quantification. (Recent explorations of intermediate representations between FOL and shallow predicate-argument relations are described in Section 14.4.)

To see how shallow semantics can represent meaning, consider these four sentences (borrowed from the slides of a tutorial by Kristina Toutanova and Scott Yih).

(14.1) [Yesterday]$_3$, [Kristina]$_0$ hit [Scott]$_1$ [with a baseball]$_2$

(14.2) [Scott]$_1$ was hit by [Kristina]$_0$ [yesterday]$_3$ [with a baseball]$_2$

[1] **This section follows closely from J&M 2009**

(14.3) [Yesterday]$_3$, [Scott]$_1$ was hit [with a baseball]$_2$ by [Kristina]$_0$

(14.4) [Kristina]$_0$ hit [Scott]$_1$ [with a baseball]$_2$ [yesterday]$_3$

We don't need first-order logic to realize that these sentences are semantically identical. Shallow semantics will suffice: the *roles* in each sentence are filled by the same text.

- [Hitter]$_0$: *Kristina*
- [Person hit]$_1$: *Scott*
- [Instrument of hitting]$_2$: *with a baseball*
- [Time of hitting]$_3$: *yesterday*

The event semantics representation for the sentence *Scott was hit by Kristina yesterday* (and all of the other examples) is:

$$\exists e. Hitting(e) \wedge Hitter(e, Kristina) \wedge PersonHit(e, Scott)$$
$$\wedge\, TimeOfHitting(e, Yesterday)$$

In this example, $Hitter$, $PersonHit$, and $TimeOfHitting$ are roles. We use these specific roles because of the **predicate verb** *hit*. Roles that relate to a specific predicate are called **deep roles**.

Thematic roles

Without knowing more about deep roles like $Hitter$, we cannot do much inference. But building classifiers for every role of every predicate would be a lot of work, and we would struggle to get enough training data to accomplish this. Is there a shortcut?

Consider the example *Scott was paid by Kristina yesterday*. Clearly *yesterday* is filling the same role in this example as in Examples section 14.1-item 14.4, describing the time at which the events occur — regardless of whether the event is *hitting* or *paying*. But arguably, the role-fillers *Scott*, *Kristina* and *yesterday* also have similar thematic functions as in the earlier sentence about baseballs.

- *Kristina* is causing the event by performing an action, which she does volitionally (on purpose); we can generalize her **thematic role** in these examples as the AGENT of the event.

- *Scott* is the primary experiencer of the effects of the event. We can generalize his thematic role as the PATIENT.

AGENT and PATIENT are the two best-known examples of **thematic roles** (Fillmore, 1968),[2] which attempt to generalize across predicates. They are also among the least controversial (Dowty, 1991); other thematic roles are shown in Table 14.1, but it is important to emphasize that this particular role inventory is not universally accepted, or even accepted to the same extent as, say, the Penn Treebank syntactic categories.

Case frames Different verbs take different thematic roles as arguments. The possible arguments for a verb is the **case frame** or **thematic grid**. For example, for *break*:

- AGENT: Subject, THEME: Object
  John broke the window.
- AGENT: Subject, THEME: Object, INSTRUMENT: PP (with)
  John broke the window with a rock.
- INSTRUMENT: Subject, THEME: Object
  The rock broke the window.
- THEME: Subject
  The window broke.

When two verbs have similar case frames, this is a clue that they might be semantically related: (e.g., *break*, *shatter*, *smash*).

Many verbs permit multiple orderings of the same arguments. These are known as **diathesis alternations**. For example, *give* permits the dative alternation,

(14.5) [$_{\text{AGENT}}$ *Doris*] ***gave*** [$_{\text{GOAL}}$ *Cary*] [$_{\text{THEME}}$ *the book*].

(14.6) [$_{\text{AGENT}}$ *Doris*] ***gave*** [$_{\text{THEME}}$ *the book*] [$_{\text{GOAL}}$ *to Cary*].

Again, similar alternation patterns suggest semantic similarity. For example, verbs that display the dative alternation include some broad classes:

- "verbs of future having" (advance, allocate, offer, owe)
- "verbs of sending" (forward, hand, mail)
- "verbs of throwing" (kick, pass, throw)

[2] The idea of thematic roles can be traced to the Sanskrit linguist Pāṇini (7th-4th century BCE!).

AGENT	The volitional causer ***The waiter** spilled the soup*
EXPERIENCER	The experiencer *The soup gave **all three of us** a headache.*
FORCE	The non-volitional causer ***The wind** blew my soup off the table.*
THEME	The participant most directly affected *The wind blew my **my soup** off the table.*
RESULT	The end product *The cook has prepared **a cold duck soup**.*
CONTENT	The proposition or content of a propositional event *The waiter assured me that **the soup is vegetarian**.*
INSTRUMENT	An instrument used in an event *It's hard to eat soup **with chopsticks**.*
BENEFICIARY	The beneficiary *The waiter brought **me** some soup.*
SOURCE	Th origin of the object of a transfer event *The stack of canned soup comes **from Pittsburgh**.*
GOAL	The destination of the object of a transfer event *He brought the bowl of soup **to our table**.*

Table 14.1: Definitions and examples of thematic roles (Jurafsky and Martin, 2009)

The purpose of thematic roles is to abstract above verb-specific roles. But it is usually possible to construct examples in which thematic roles are insufficiently specific.

- *Intermediary instruments* can act as subjects:

  1. *The cook opened the jar with the new gadget.*
  2. *The new gadget opened the jar.*

- *Enabling instruments* cannot:

  1. *Shelly ate the pizza with the fork.*

2. **The fork ate the pizza.*

Thematic roles are bundles of semantic properties, but it's not clear how many properties are necessary. For example, AGENTS are usually animate, volitional, sentient, causal, but any of these properties may be missing occasionally. The distinction between agents and patients is explored in detail by Dowty (1991).

The Proposition Bank

In the Proposition Bank (**PropBank**), roles are verb-specific, with some sharing (Palmer et al., 2005).

- ARG0: proto-agent (has agent-like properties)
- ARG1: proto-patient (has patient-like properties)
- ARG2 ... ARGN: verb-specific
- 13 universal adjunct-like arguments: temporal, manner, location, cause, negation, ...

PropBank contains two main resources:[4] "frame files" describing the roles for each verbal predicate (3,324 such files are included), and labeled sentences, built on the Penn TreeBank (113,000 such propositions are annotated). Some example PropBank-style sentence annotations are shown in Figure 14.1. The overlap with the Penn Treebank makes it possible to test the relationship between semantic roles and syntactic constituents. Similar PropBanks have been created for other languages, including Arabic, Chinese, Hindi, and Korean. PropBank is used as the standard dataset for popular shared tasks on Semantic Role Labeling (SRL); some of the main approaches are described in section 14.2.

PropBank describes the predicate-argument structure of verbs, but words belong to other syntactic categories may have argument structures of their own. A related resource is NomBank (Meyers et al., 2004), which annotates the arguments of noun phrases, such as:

(14.7) [$_{\text{ARG0}}$ students'] [$_{\text{REL}}$ knowledge] *of* [$_{\text{ARG1}}$ two-letter consonant sounds]

In this example, the syntactic head is *knowledge*, and this is also the word that defines the semantic relation (REL). The "proto-agent" in this case is *students'*, and the "proto-patient" is *two-letter consonant sounds*.

[4] https://catalog.ldc.upenn.edu/LDC2004T14; http://verbs.colorado.edu/propbank/framesets-english/scratch-v.html

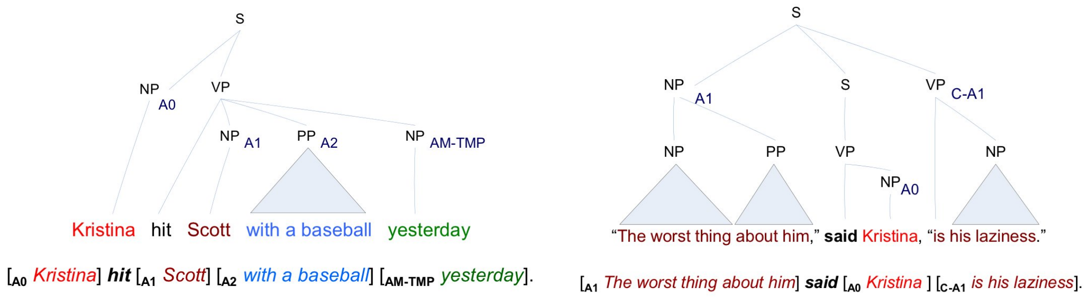

S
NP A0
VP
NP A1
PP A2
NP AM-TMP
Kristina hit Scott with a baseball yesterday
[A0 Kristina] hit [A1 Scott] [A2 with a baseball] [AM-TMP yesterday].
S
NP A1
S
VP C-A1
NP
PP
VP
NP A0
NP
"The worst thing about him," said Kristina, "is his laziness."
[A1 The worst thing about him] said [A0 Kristina] [C-A1 is his laziness].

Figure 14.1: Examples of PropBank-style annotations, borrowed from the slides of Toutanova and Yih

14.2 Semantic Role Labeling

Semantic role labeling (SRL) is the task of assigning semantic labels to spans of text. Labels describe the role of the phrase with respect to the *predicate verb*. In practice, this usually means PropBank labels, e.g. Arg0, Arg1, etc, so our goal is to produce labelings such as those shown in Figure 14.1.

While there are many possible approaches to Semantic Role Labeling (SRL), an effective solution is to treat it as another case of structured prediction. The problem has a few components:

1. identify all predicates in the sentence;
2. identify all argument spans;
3. label the argument spans.

Early approaches treated these problems in isolation, but more recent work has shown that it is best to treat them jointly. Assuming for the moment that we have identified the predicate, the remaining problem can be viewed as simply **tagging** the remaining words in a tagset $\mathcal{T} = \{\text{A0}, \text{A1}, \text{A2}, \ldots, \text{AM-TMP}, \ldots, \varnothing\}$. Thus, the output of an SRL system might be written,

(14.8) *Kristina*/A0 *hit*/PRED *Scott*/A1 *with*/A2 *a*/A2 *baseball*/A2

This would suggest that SRL can be solved by applying a sequence labeling algorithm such as structured perceptron with Viterbi. But recall that Viterbi is based on sequential features, $\boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}) = \sum_m \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m)$; these features are not particularly useful in SRL, because sequential constraints and preferences are less important here than they are in tasks such as part-of-speech tagging and named-entity recognition — recall examples (section 14.1-item 14.4). In fact, it is better to

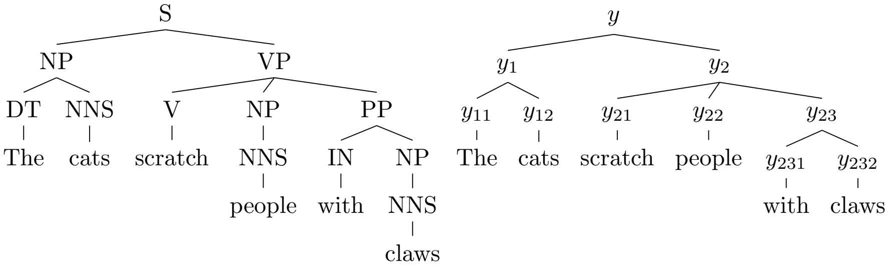


Figure 14.2: Conversion of a constituent parse tree to variables for semantic role labeling

consider the tree structure offered by a constituent parse of the sentence: in PropBank, 96% of the arguments correspond to a "gold" constituent (from the manaul annotation), and 90% correspond to a constituent from an automatic parser (Punyakanok et al., 2008). Therefore we will treat the problem of SRL as a problem of **labeling constituents**, rather than labeling words. This transformation is illustrated in Figure 14.2.

Given a sentence $\boldsymbol{w}$ and a parse tree τ, our goal is now to assign each y_i to a value in the set $\mathcal{T}$. We optimize a scoring function,

$$\hat{\boldsymbol{y}} = \text{argmax}\, \boldsymbol{y} \in \mathcal{Y}(\boldsymbol{w}, \tau) \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{y}, \boldsymbol{w}, \tau) \tag{14.1}$$

$$\boldsymbol{f}(\boldsymbol{y}, \boldsymbol{w}, \tau) = \sum_{i}^{\text{constituents}(\tau)} \boldsymbol{f}(y_i, \boldsymbol{w}, \tau), \tag{14.2}$$

where we assume that the features decompose across labels y_i. Notice that the features may consider any part of the parse tree, since we are not searching over parse trees. Useful features for this problem include: the predicate verb (which is given); the syntactic type (e.g., NP, VP), head word, first word, and last word of the constituent; whether the constituents comes before or after the predicate; and the **syntactic path** from the constituent to the predicate. This last feature describes a series of steps up and down the parse tree: in the example shown in Figure 14.2, the path from *the cats* (y_1) to the predicate *scratch* (y_{21}) is written NP $\uparrow$ S $\downarrow$ VP $\downarrow$ V. The syntactic path feature captures regularities in the syntactic positions of constituent arguments. For more discussion of features, see Gildea and Jurafsky (2002) and Surdeanu et al. (2007).

The inference problem defined in Equation 14.1 specifies a search over $\mathcal{Y}(\boldsymbol{w}, \tau)$, which is all permissible labelings of the parse tree τ for the sentence $\boldsymbol{w}$. How should we define this set? If every constituent is allowed to have any label in $\mathcal{T}$, then we have $\mathcal{Y}(\boldsymbol{w}, \tau) = \mathcal{T}^{\#|\tau|}$. But this seems too permissive: it would allow a single argument to appear in multiple places (for example, both *cats* and *claws* labeled as $A0$), and would also allow multi-word constituents like *the cats* to realize a different argument from their children, like *cats*.

Rather than explicitly defining the set $\mathcal{Y}(\boldsymbol{w}, \tau)$, it is useful to think of **constraints** that a labeling $\boldsymbol{y}$ must obey. To do this, we will redefine $\boldsymbol{y}$ slightly, so that it includes a set of indicator features,

$$Y_{i,t} = \begin{cases} 1, & \text{argument } i \text{ takes tag } t \\ 0, & \text{otherwise} \end{cases} \tag{14.3}$$

Now, we can define $\mathcal{Y}(\boldsymbol{w}, \tau)$ to include only those labelings that obey a set of **contraints**. For example:

- All arguments get at most one label, $\forall i \sum_t y_{i,t} = 1$. Note we use equality, because you can always have the $\varnothing$ label.
- No duplicate argument classes, $\forall t \neq \varnothing, \sum_i y_{i,t} \leq 1$
- Overlapping arguments get at most one non-null label:

$$\forall \langle i, j \rangle : i \rightsquigarrow_\tau j, y_{i,\varnothing} + y_{j,\varnothing} \geq 1 \tag{14.4}$$

- Some arguments are forbidden, e.g. $\sum_i y_{i,A2} = 0$. Many predicates cannot take all types of arguments: for example, the verb *dream* can only take $A0$ and $A1$, so we would add this contraint to make it impossible to label anything as $A2$ or $A3$.

All of the constraints are linear, meaning we can write them as a matrix-vector product, $\mathbf{A}\boldsymbol{y} \leq \boldsymbol{b}$. Moreover, we can redefine the feature function as $\boldsymbol{f}(y_i, \boldsymbol{w}, \tau, i) = \sum_t y_{i,t} \times \boldsymbol{f}(\boldsymbol{w}, \tau, i, t)$, so that the scoring function is,

$$\begin{aligned} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{y}, \boldsymbol{w}, \tau) = & \sum_i \boldsymbol{\theta}^\top \boldsymbol{f}(y_i, \boldsymbol{w}, \tau, i) && (14.5) \\ = & \sum_i \sum_t (\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, \tau, i, t)) \times y_{i,t}. && (14.6) \end{aligned}$$

We can therefore reframe the overall optimization problem as,

$$\begin{aligned} \hat{\boldsymbol{y}} = & \text{argmax}\, \boldsymbol{y} \in \mathcal{T}^{\#|\tau|} \sum_i \sum_t (\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, \tau, i, t)) \times y_{i,t} && (14.7) \\ s.t. \mathbf{A}\boldsymbol{y} \leq \boldsymbol{b} & && (14.8) \end{aligned}$$

The objective function is linear in $\boldsymbol{y}$, the constraints are linear inequalities, and each $Y_{i,t} \in \{0, 1\}$. This optimization problem is therefore a case of **integer linear programming** (ILP). Unfortunately, ILP is known to be NL-hard, including in the binary special case. However, because ILP has many commercial applications, it is a well-studied problem, with heuristic approximations that work well in the overwhelming majority of practical cases. One such algorithm is implemented in the free software GNU Linear Programming Kit (GLPK); Gurobi and CPLEX provide commercial implementations. Integer linear programming is an example of a **combinatorial optimization** problem, with alternative solutions such as **dual decomposition**. Das et al. (2012) develop an "augmented" dual decomposition algorithm which obtains identical accuracy to CPLEX, while running roughly ten times faster.

A final note about constrained optimization approaches to SRL is that you might be uncomfortable about committing to a single syntactic parse, given that even the best parsers have a 10% error rate. Punyakanok et al. (2008) show that you can do better by considering the constituents of five different parsers at the same time! The trick is simple: add constraints preventing the optimizer from selecting constituents that overlap across parses.

Applications of SRL Why might we want to do this? One application is to automatic question answering systems like IBM Watson. Consider the example question, *Who discovered prions?*. Somewhere in our database, we have the statement *1997: Stanley B. Prusiner, United States, discovery of prions....* How can we link them up? Shen and Lapata (2007) use semantic roles to align questions against the content of factual sentences, as shown in Figure 14.3.

[todo: more applications]

14.3 FrameNet

PropBank does not attempt to group related predicates, such as BUY/SELL, GIVE/RECEIVE, and RISE/FALL. FrameNet provides a richer model of shallow semantics by grouping predicates and arguments into a predefined **frame** ontology. To see how this works, consider the following examples from Jurafsky and Martin (2009):

(14.9) [$_{\text{A1}}$ *The price of bananas*] *increased* [$_{\text{A2}}$ *5%*].

(14.10) [$_{\text{A1}}$ *The price of bananas*] *rose* [$_{\text{A2}}$ *5%*].

(14.11) *There has been a* [$_{\text{A2}}$ *5%*] *increase* [$_{\text{A1}}$ *in the price of bananas*].

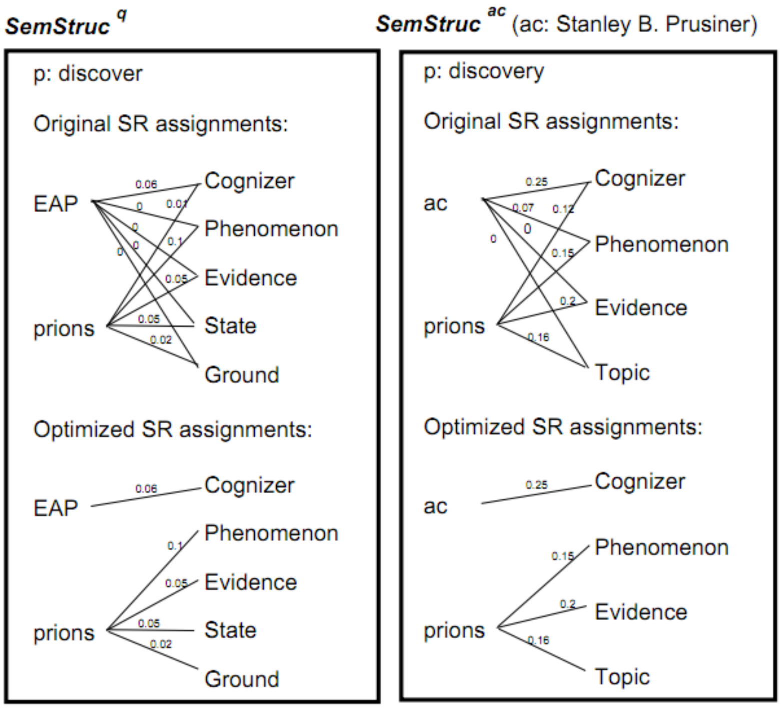


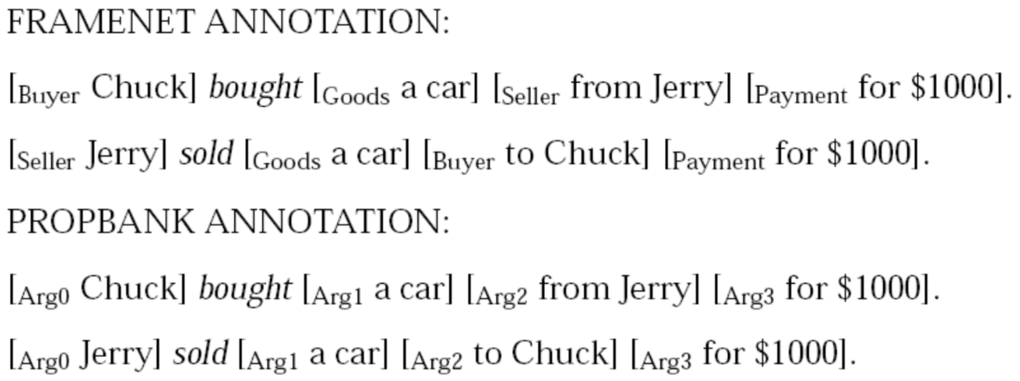
Figure 14.3: Using semantic role labeling to align questions and answers

The first two sentences involve different verbs; the second sentence conveys same semantics with a noun. Nonetheless, the meaning is the same.

A frame defines a set of *lexical units* and a set of *frame elements*, as shown in Figure 14.5. The relationship between Framenet and PropBank annotation is shown in Figure 14.4. The FrameNet corpus is publicly available online,[6], and of this writing, annotation is still ongoing.

Unlike PropBank, Framenet is not based on TreeBank parses, and example sentences are chosen by hand. Shi and Mihalcea (2004) present a deterministic algorithm for FrameNet parsing, and Das et al. (2010, 2014) provide a structured prediction approach. But compared to PropBank, there is much less work on parsing to the Framenet representation.

[6] https://framenet.icsi.berkeley.edu/fndrupal/about



FRAMENET ANNOTATION:

[$_{\text{Buyer}}$ Chuck] *bought* [$_{\text{Goods}}$ a car] [$_{\text{Seller}}$ from Jerry] [$_{\text{Payment}}$ for $1000].

[$_{\text{Seller}}$ Jerry] *sold* [$_{\text{Goods}}$ a car] [$_{\text{Buyer}}$ to Chuck] [$_{\text{Payment}}$ for $1000].

PROPBANK ANNOTATION:

[$_{\text{Arg0}}$ Chuck] *bought* [$_{\text{Arg1}}$ a car] [$_{\text{Arg2}}$ from Jerry] [$_{\text{Arg3}}$ for $1000].

[$_{\text{Arg0}}$ Jerry] *sold* [$_{\text{Arg1}}$ a car] [$_{\text{Arg2}}$ to Chuck] [$_{\text{Arg3}}$ for $1000].

Figure 14.4: A comparison of framenet and propbank, from Toutanova and Yih [todo: I think]

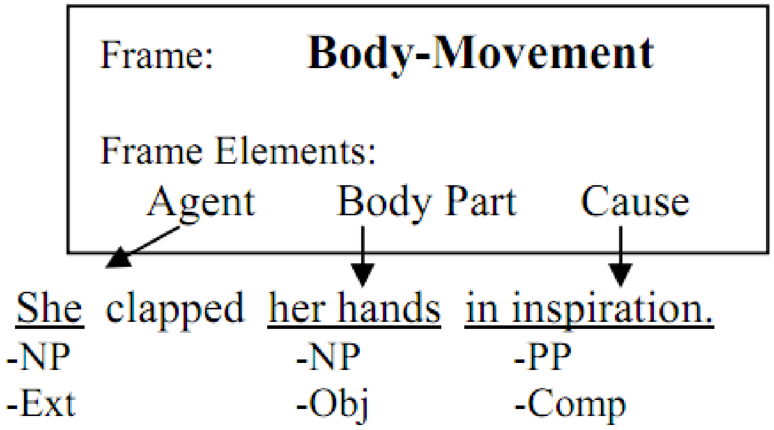


Figure 14.5: FrameNet annotation, figure from Fleischman et al, 2003

14.4 Abstract Meaning Representation

Recent work has focused on a new form of shallow semantics, the **abstract meaning representation** (AMR), which is more structured than PropBank-style semantics, but less ambitious than first-order logic. A major gap in semantic role labeling is the inability to link arguments that refer to a single entity: for example:

(14.12) Abby told Max she would visit him in San Quentin.

In this example, there are three entities, *Abby*, *Max*, and *San Quentin*, and two predicates, *told* and *visit*. The associated AMR structure is shown in Figure 14.6.

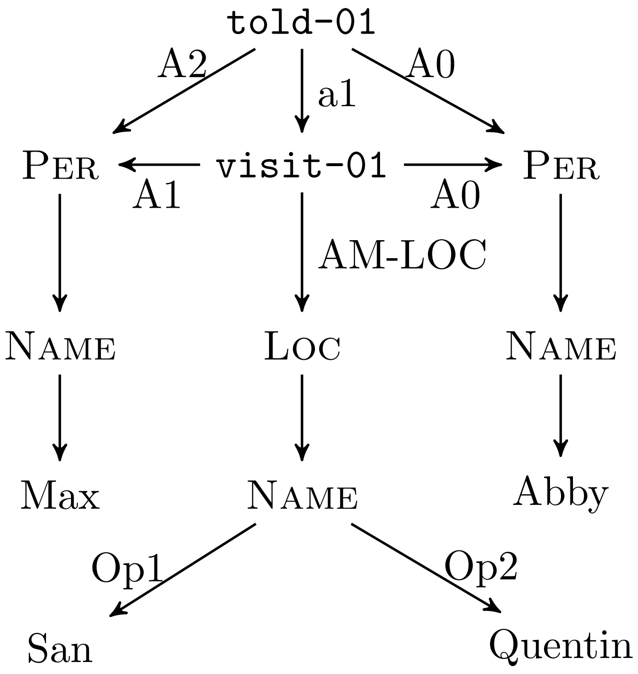


Figure 14.6: An example parse in the Abstract Meaning Representation (AMR)

This graph includes a number of pieces of information about the semantics. The **coreference** relations between *Abby* and *she*, and *Max* and *he* are indicated by having multiple incoming arrows to the nodes representing these entities. In addition, the types of the entities are represented with special nodes: *Abby* and *Max* are of type PER, person; *San Quentin* is of type LOC. The graph also indicates the sense of each predicate, the relationship between the predicates, and the role of each argument.

[todo: Talk a little about AMR parsing] [todo: talk about applications of AMR]

Chapter 15

Distributional and distributed semantics

A recurring theme in this course is that the mapping from words to meaning is complex.

Word sense disambiguation A single form, like *bank*, may have multiple meanings.

Synonymy Conversely, a single meaning may be created by multiple surface forms, as represented by the *synsets* described in section 3.2

Paradigmatic relations Other lexical semantic relationships include antonymy (opposite meaning), hyponymy (instance-of), and meronymy (part-whole)

Moreover, both compositional and frame semantics assume hand-crafted **lexicons** that map from words to predicates. But how can we do semantic analysis of words that we've never seen before?

15.1 The distributional hypothesis

Here's a word you may not know: *tezgüino*. If we encounter this word, what can we do? It seems like a big problem for any NLP system, from POS tagging to semantic analysis.

Suppose we see that *tezgüino* is used in the following contexts:[1]

[1] Example from Lin (1998).

(15.1) *A bottle of* __________ *is on the table.*

(15.2) *Everybody likes* __________*.*

(15.3) *Don't have* __________ *before you drive.*

(15.4) *We make* __________ *out of corn.*

What other words fit into these contexts? How about: *loud, motor oil, tortillas, choices, wine*? We can create a vector for each word, based on whether it can be used in each context.

	C1	C2	C3	C4	...
tezgüino	1	1	1	1	
loud	0	0	0	0	
motor oil	1	0	0	1	
tortillas	0	1	0	1	
choices	0	1	0	0	
wine	1	1	1	1	

Based on these vectors, it seems that:

- *wine* is very similar to *tezgüino*;
- *motor oil* and *tortillas* are fairly similar to *tezgüino*;
- *loud* is quite different.

The vectors describe the **distributional** properties of each word. Does vector similarity imply semantic similarity? This is the **distributional hypothesis**, stated by Firth (1957) as: "You shall know a word by the company it keeps." It is also known as a **vector-space model**, since each word's meaning is captured by a vector. Vector-space models and distributional semantics are relevant to a wide range of NLP applications.

Query expansion search for *bike*, match *bicycle*;

Semi-supervised learning use large unlabeled datasets to acquire features that are useful in supervised learning;

Lexicon and thesaurus induction automatically expand hand-crafted lexical resources, or induce them from raw text.

Vector-space models typically fill out the vector representation using contextual information about each word, known as **distributional statistics**. In the example above, the vectors are composed of binary values, indicating whether it is conceptually possible for a word to appear in each context. But in real systems, we will compute distributional statistics from corpora, using various definitions of context. This definition can have a major impact on the lexical semantics that results; for example, Marco Baroni (lecture slides) computes the thirty nearest neighbors of the word *dog*, based on the counts of all words that appear within a fixed window of the target word. Varying the size of the window yields quite different results:

2-word window *cat, horse, fox, pet, rabbit, pig, animal, mongrel, sheep, pigeon*

30-word window *kennel, puppy, pet, bitch, terrier, rottweiler, canine, cat, (to) bark, Alsatian*

Each word in the two-word window is an animal, reflecting the fact that locally, the word *dog* tends to appear in the same contexts as other animal types (e.g., *pet the dog*, *feed the dog*, etc). In the 30-word window, nearly everything is dog-related, including specific breeds such as *rottweiler* and *Alsatian*, but the list also includes words that are not animals (*kennel*), and in one case (*bark*), is not a noun at all. The reason is that the 2-word window is more sensitive to syntax, while the 30-word window is more sensitive to topic.

15.2 Distributional semantics

Local distributional statistics: Brown clusters

One way to use context is to perform word clustering. This can improve the performance of downstream (supervised learning) tasks, because even if a word is not observed in any labeled instances, other members of its clusters might be. The Brown et al. (1992) clustering algorithm provides one way to do this. The algorithm is over 20 years old and is still widely used in NLP; for example, Owoputi et al. (2012) use it to obtain large improvements in Twitter part-of-speech tagging.[2]

In Brown clustering, the context is just the immediately adjacent words. The similarity metric is built on a generative probability model:

[2] You can download Brown clusters at `http://metaoptimize.com/projects/wordreprs/`.

Algorithm 9 The bottom-up Brown et al. (1992) clustering algorithm

```
$\forall w, C(w) = w$ (start with every word in its own cluster)
while all clusters not merged do
    merge the $c_i$ and $c_j$ to maximize clustering quality.
Each word is described by a bitstring representation of its merge path
```

- Assume each word w has a class $C(w)$
- Assume a generative model $\log \text{p}(w) = \sum_i \log \text{p}(w_i \mid c_i) + \log \text{p}(c_i \mid c_{i-1})$ (What does this remind you of?)

The word clusters $C(w)$ are not observed; our goal is to infer them from data. Now, in this model, we assume that,

$$\text{p}(w_i \mid c_i) = \begin{cases} \frac{\text{count}(w_i)}{\text{count}(c_i)}, & c_i = C(w_i) \\ 0, & \text{otherwise.} \end{cases} \tag{15.1}$$

This means that each word **type** has a single cluster — unlike in hidden Markov models, where a given word might be generated from multiple tags. Due to this constraint, we will not apply the expectation maximization algorithm which was used in unsupervised hidden markov model learning (section 6.6). Instead, Brown et al. (1992) use a hierarchical clustering algorithm, shown in Algorithm 9. This is a **bottom-up** clustering algorithm, in that every word begins in its own cluster, and then clusters are merged until everything is clustered together. The series of merges taken by the algorithm is called a **dendrogram**, and it looks like a tree. For example, if the words *bike* and *bicycle* are first merged with each other, and then the cluster was merged with another cluster containing just the word *tricycle*, we would have the small tree shown in Figure 15.1.

For any desired number of clusters K, we can get a clustering by "cutting" the tree at some height. But in Brown clustering, we are usually interested not only in the resulting clusters from some cut of the merge tree, but also in the bitstrings that represent the series of mergers that led to the final clustering. A classical approach to semi-supervised learning is to use Brown bitstring prefixes in place of (or in addition to) lexical features, thus generalizing to words that are unseen in labeled data. The bitstrings for Figure 15.1 would be 0 for *tricycle*, 10 for *bicycle*, and 11 for *bike*. Subtrees from Brown clustering on a larger dataset are shown in Figure 15.2. The examples are drawn from a paper by Miller et al. (2004), who use Brown cluster bitstring prefixes as features for named entity recognition;

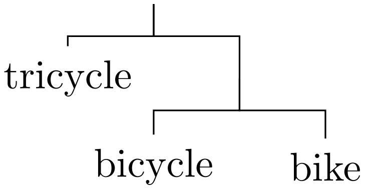


Figure 15.1: A small subtree produced by bottom-up Brown clustering

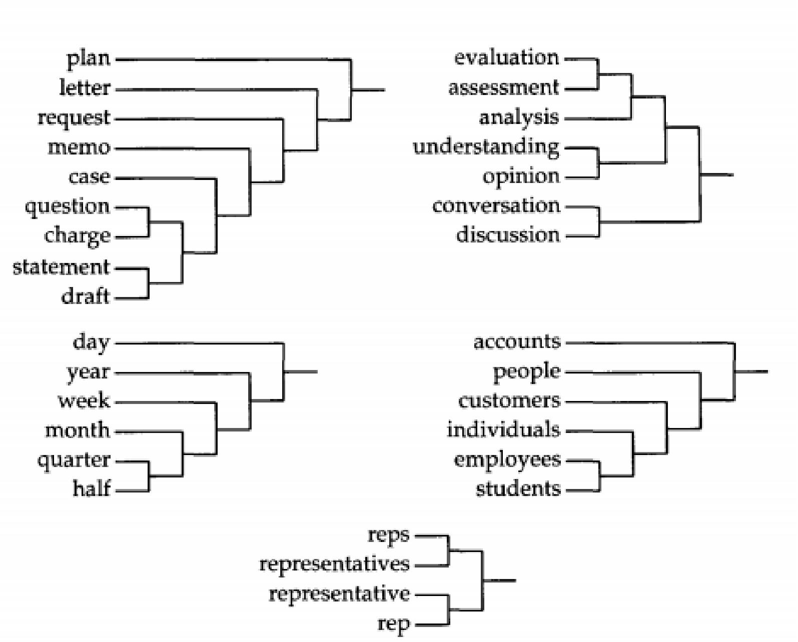


Figure 15.2: Brown subtrees from Miller et al. (2004)

this approach has also been used in dependency parsing (Koo et al., 2008) and in Twitter part-of-speech tagging (Owoputi et al., 2012).

The complexity of Algorithm 9 is $\mathcal{O}(V^3)$, where V is the size of the vocabulary. We are merging V clusters, since we start off with each word in its own cluster; each merger involves searching over $\mathcal{O}(V^2)$ pairs of clusters, to find the pair that maximizes the improvement in clustering quality. Cubic complexity is too slow for practical purposes, so we will explore a faster approximate algorithm later.

Brown clusters and mutual information

We now explore the Brown clustering algorithm more mathematically, and then derive a more efficient clustering algorithm. First, some notation:

- $\mathcal{V}$ is the set of all words.
- N is number of observed word tokens.
- $C : \mathcal{V} \to \{1, 2, \ldots, k\}$ defines a partition of words into k classes.
- count(w) is the number of times we see word $w \in \mathcal{V}$. This function can also be used to count classes.
- count(w, v) is the number of times w immediately precedes v. This function can also be used to count class bigrams.

$$\begin{aligned}
\mathrm{p}(w_1, w_2, \ldots, w_N; C) &= \prod_m \mathrm{p}(w_m \mid C(w_m))\mathrm{p}(C(w_m) \mid C(w_{m-1})) \\
\log \mathrm{p}(w_1, w_2, \ldots, w_N; C) &= \sum_m \log \mathrm{p}(w_m \mid C(w_m)) \times \mathrm{p}(C(w_m) \mid C(w_{m-1}))
\end{aligned}$$

This is kind of like a hidden Markov model, but each word can only be produced by a single cluster. Now let's define the "quality" of a clustering as the average log-likelihood:

$$
\begin{aligned}
J(C) =& \frac{1}{N}\sum_{m}^{N} \log \left(\mathrm{p}(w_m \mid C(w_m)) \times \mathrm{p}(C(w_m) \mid C(w_{m-1}))\right) && \\
=& \sum_{w,w'} \frac{n(w,w')}{N} \log \left(\mathrm{p}(w' \mid C(w')) \times \mathrm{p}(C(w') \mid C(w'))\right) && \text{sum over word types instead} \\
=& \sum_{w,w'} \frac{n(w,w')}{N} \log \left(\frac{n(w')}{n(C(w'))} \times \frac{n(C(w),C(w'))}{n(C(w))}\right) && \text{definition of probabilities} \\
=& \sum_{w,w'} \frac{n(w,w')}{N} \log \left(\frac{n(w')}{1} \times \frac{n(C(w),C(w'))}{n(C(w)) \times n(C(w'))} \times \frac{N}{N}\right) && \text{re-arrange, multiply by one} \\
=& \sum_{w,w'} \frac{n(w,w')}{N} \log \left(\frac{n(w')}{N} \times \frac{n(C(w),C(w')) \times N}{n(C(w)) \times n(C(w'))}\right) && \text{re-arrange terms} \\
=& \sum_{w,w'} \frac{n(w,w')}{N} \log \frac{n(w')}{N} + \frac{n(w,w')}{N} \log \left(\frac{n(C(w),C(w')) \times N}{n(C(w)) \times n(C(w'))}\right) && \text{distribution through log} \\
=& \sum_{w'} \frac{n(w')}{N} \log \frac{n(w')}{N} + \sum_{c,c'} \frac{n(c,c')}{N} \log \left(\frac{n(c,c') \times N}{n(c) \times n(c')}\right) && \text{sum across bigrams and classes} \\
=& \sum_{w'} \mathrm{p}(w') \log \mathrm{p}(w') + \sum_{c,c'} \mathrm{p}(c,c') \log \frac{\mathrm{p}(c,c')}{\mathrm{p}(c) \times \mathrm{p}(c')} && \text{multiply by } \frac{N^{-2}}{N^{-2}} \text{ inside log} \\
=& -H(W) + I(C) &&
\end{aligned}
$$

The last step uses the following definitions from information theory:

Entropy The entropy of a discrete random variable is the expected negative log-likelihood,

$$H(X) = -E[\log P(X)] = -\sum_{x} P(X = x) \log P(X = x). \tag{15.2}$$

For example, for a fair coin we have $H(X) = \frac{1}{2} \log \frac{1}{2} + \frac{1}{2} \log \frac{1}{2} = -\log 2$; for a (virtually) certain outcome, we have $H(x) = 1 \times \log 1 + 0 \times \log 0 = 0$. We have already seen entropy in a few other contexts.

Mutual information The information shared by two random variables is the mutual information,

$$I(X;Y) = \sum_{y \in Y} \sum_{x \in X} \mathrm{p}_{X,Y}(x,y) \log \left(\frac{\mathrm{p}_{X,Y}(x,y)}{\mathrm{p}_X(x)\mathrm{p}_Y(y)}\right). \tag{15.3}$$

Algorithm 10 Exchange clustering algorithm

```
For $K$ most frequent words, set $C_i = i$.
for $i = (m+1) : V$ do
    Set $C_i = K + 1$
    Let $\langle c, c' \rangle$ be the two clusters whose merger minimizes the decrease in $I(C)$
    Merge $c$ and $c'$
```

> For example, if X and Y are independent, then $\mathrm{p}_{X,Y}(x,y) = \mathrm{p}_X(x)\mathrm{p}_Y(y)$, so the mutual information is $\log 1 = 0$. In

By $I(C)$, we are using a shorthand for the mutual information of adjacent word classes, $\langle C_{m-1}, C_m \rangle$,

$$I(C) = \sum_{C_m = c, C_{m-1} = c'} P(C_m = c, C_{m-1} = c') \log \left(\frac{P(C_m = c, C_{m-1} = c')}{P(C_m = c) P(C_{m-1} = c')} \right) \quad (15.4)$$

The entropy $H(W)$ does not depend on the clustering, so this term is constant; choosing a clustering with maximum mutual information $I(C)$ is equivalent to maximizing the log-likelihood. Now let's see how to do that efficiently.

$V \log V$ **approximate algorithm**

With this model in hand, we can now define a more efficient algorithm, shown in Algorithm 10. The algorithm keeps exactly K clusters at every point in time, so the merger operation requires considering only $\mathcal{O}(K^2)$ clusters. We have to pass over the entire vocabulary once for a cost of $\mathcal{O}(V)$, but more importantly, we must sort the words by frequency, for a cost of $\mathcal{O}(V \log V)$, giving a total cost of $\mathcal{O}(V \log V + VK^2)$.

Syntactic distributional statistics

Local context is contingent on syntactic decisions that may have little to do with semantics:

(15.5) *I gave Tim the ball.*

(15.6) *I gave the ball to Tim.*

(You may recall from section 14.1 that this is the **dative alternation**.) Using the syntactic structure of the sentence might give us a more meaningful context, yielding better clusters.

There are several examples of this idea in practice. Pereira et al. (1993) cluster nouns based on the verbs for which they are the direct object: the context vector for each noun is **the count of occurences as a direct object of each verb**. As with Brown clustering, they employ a class-based probability model:

$$\hat{\mathrm{p}}(n, v) = \sum_{c \in \mathcal{C}} \mathrm{p}(v \mid c) \times \mathrm{p}(c, n) \tag{15.5}$$

$$= \sum_{c \in \mathcal{C}} \mathrm{p}(v \mid c) \times \mathrm{p}(n \mid c) \times \mathrm{p}(c), \tag{15.6}$$

where n is the noun, v is the verb, and c is the class of the noun. They maximize the likelihood under this model using an iterative algorithm similar to expectation maximization (chapter 4).

Lin (1998) extends this idea from nouns to all words, using context statistics based on the incoming dependency edges. For any pair of words i and j and relation r, we can compute:

$$\mathrm{p}(i, j \mid r) = \frac{n(i, j, r)}{\sum_{i', j'} n(i', j', r)} \tag{15.7}$$

$$\mathrm{p}(i \mid r) = \sum_{j} \mathrm{p}(i, j \mid r) \tag{15.8}$$

Now, let $T(i)$ be the set of pairs $\langle j, r \rangle$ such that $\mathrm{p}(i, j \mid r) > \mathrm{p}(i \mid r) \times \mathrm{p}(j \mid r)$: then $T(i)$ contains words j that are especially likely to be joined with word i in relation r. Similarity between u and v can be defined through $T(u)$ and $T(v)$.

Lin considers several similarity measures for $T(u)$ and $T(v)$. Many of these are used widely in other contexts (usually for comparing clusterings or other sets), and are worth knowing about:

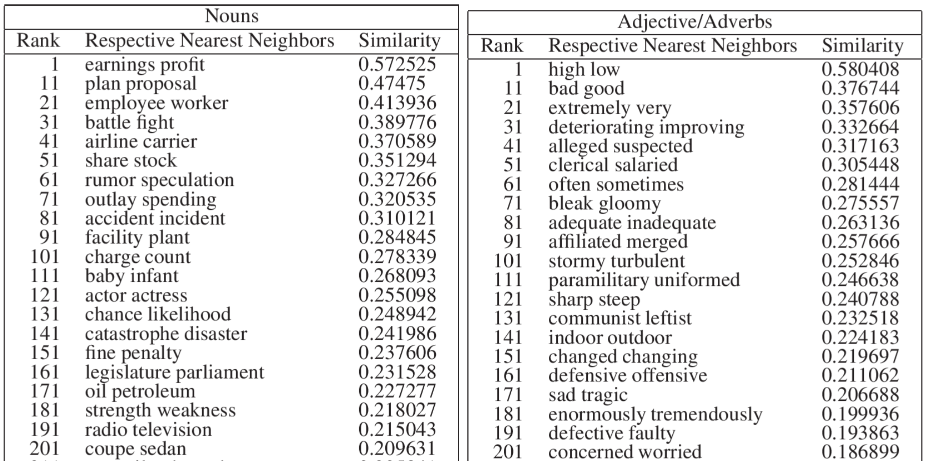
Cosine similarity $\frac{|T(u) \cap T(v)|}{\sqrt{|T(u)||T(v)|}}$

Dice similarity $\frac{2 \times |T(u) \cap T(v)|}{|T(u)| + |T(v)|}$

Jaccard similarity $\frac{|T(u) \cap T(v)|}{|T(u)| + |T(v)| - |T(u) \cap T(v)|}$

However, Lin's chosen metric is more complex than any of these well-known alternatives:

$$\frac{\sum_{\langle r, w \rangle \in T(u) \cup T(v)} I(u, r, w) + I(v, r, w)}{\sum_{\langle r, w \rangle \in T(u)} I(u, r, w) + \sum_{\langle r, w \rangle \in T(v)} I(v, r, w)}, \tag{15.9}$$



Nouns		
Rank	Respective Nearest Neighbors	Similarity
1	earnings profit	0.572525
11	plan proposal	0.47475
21	employee worker	0.413936
31	battle fight	0.389776
41	airline carrier	0.370589
51	share stock	0.351294
61	rumor speculation	0.327266
71	outlay spending	0.320535
81	accident incident	0.310121
91	facility plant	0.284845
101	charge count	0.278339
111	baby infant	0.268093
121	actor actress	0.255098
131	chance likelihood	0.248942
141	catastrophe disaster	0.241986
151	fine penalty	0.237606
161	legislature parliament	0.231528
171	oil petroleum	0.227277
181	strength weakness	0.218027
191	radio television	0.215043
201	coupe sedan	0.209631

Adjective/Adverbs		
Rank	Respective Nearest Neighbors	Similarity
1	high low	0.580408
11	bad good	0.376744
21	extremely very	0.357606
31	deteriorating improving	0.332664
41	alleged suspected	0.317163
51	clerical salaried	0.305448
61	often sometimes	0.281444
71	bleak gloomy	0.275557
81	adequate inadequate	0.263136
91	affiliated merged	0.257666
101	stormy turbulent	0.252846
111	paramilitary uniformed	0.246638
121	sharp steep	0.240788
131	communist leftist	0.232518
141	indoor outdoor	0.224183
151	changed changing	0.219697
161	defensive offensive	0.211062
171	sad tragic	0.206688
181	enormously tremendously	0.199936
191	defective faulty	0.193863
201	concerned worried	0.186899

Figure 15.3: Similar word pairs from the clustering method of Lin (1998)

where $I(u, r, w)$ is the mutual information between u and w, conditioned on r.

Results of the algorithm are shown in Figure 15.3. An interesting point in these results is that while many of the pairs are indeed synonyms, some have the **opposite** meaning. This is particularly evident for the adjectives, with pairs like *good/bad* and *high/low* at the top. It's useful to think about why this might be the case, and how you might fix it.

Lin's algorithm was also evaluated on its ability to match synonym pairs in human-generated thesauri. Its measure of text similarity was a better matched to WordNet than was the (human-written) Roget thesaurus!

15.3 Distributed representations

Distributional semantics are computed from context statistics. **Distributed** semantics are a related but distinct idea: that meaning is best represented by numerical vectors rather than discrete combinatoric structures. Distributed representations are often distributional: this section will focus on latent semantic analysis and word2vec, both of which are distributed representations that are based on distributional statistics. However, distributed representations need not be distributional: for example, they can be learned in a supervised fashion from labeled data, as in the sentiment analysis work of Socher et al. (2013b).

Latent semantic analysis

Thus far, we have considered context vectors that are large and sparse. We can arrange these vectors into a matrix $\mathbf{X} \in \mathbb{R}^{V \times N}$, where rows correspond to words and columns correspond to contexts. However, for rare words i and j, we might have $\boldsymbol{x}_i^\top \boldsymbol{x}_j = 0$, indicating zero counts of shared contexts. So we'd like to have a more robust representation.

We can obtain this by factoring $\mathbf{X} \approx \mathbf{U}_K \mathbf{S}_K \mathbf{V}_K^\top$, where

$$\mathbf{U}_K \in \mathbb{R}^{V \times K}, \qquad \mathbf{U}_K \mathbf{U}_K^\top = \mathbb{I} \tag{15.10}$$

$$\mathbf{S}_K \in \mathbb{R}^{K \times K}, \qquad \mathbf{S}_K \text{ is diagonal, non-negative} \tag{15.11}$$

$$\mathbf{V}_K \in \mathbb{R}^{D \times K}, \qquad \mathbf{V}_K \mathbf{V}_K^\top = \mathbb{I} \tag{15.12}$$

Here K is a parameter that determines the fidelity of the factorization; if $K = \min(V, N)$, then $\mathbf{X} = \mathbf{U}_K \mathbf{S}_K \mathbf{V}_K^\top$. Otherwise, we have

$$\mathbf{U}_K, \mathbf{S}_K, \mathbf{V}_K = \operatorname{argmin} \mathbf{U}_k, \mathbf{S}_K, \mathbf{V}_K ||\mathbf{X} - \mathbf{U}_K \mathbf{S}_K \mathbf{V}_K^\top||_F, \tag{15.13}$$

subject to the constraints above. This means that $\mathbf{U}_K, \mathbf{S}_K, \mathbf{V}_K$ give the rank-K matrix $\tilde{\mathbf{X}}$ that minimizes the Frobenius norm, $\sqrt{\sum_{i,j} (x_{i,j} - \tilde{x}_{i,j})^2}$.

This factorization is called the **Truncated Singular Value Decomposition**, and is closely related to eigenvalue decomposition of the matrices $\mathbf{X}\mathbf{X}^\top$ and $\mathbf{X}^\top\mathbf{X}$. In general, the complexity of SVD is $\min\left(\mathcal{O}(D^2 V), \mathcal{O}(V^2 N)\right)$. The standard library LAPACK (Linear Algebra PACKage) includes an iterative optimization solution for SVD, and (I think) this what is called by Matlab and Numpy.

However, for large sparse matrices it is often more efficient to take a stochastic gradient approach. Each word-context observation $\langle w, c \rangle$ gives a gradient on $\boldsymbol{u}_w$, $\boldsymbol{v}_c$, and $\boldsymbol{S}$, so we can take a gradient step. This is part of the algorithm that was used to win the Netflix challenge for predicting movie recommendation — in that case, the matrix includes raters and movies (Koren et al., 2009).

Return to NLP applications, the slides provide a nice example from Deerwester et al. (1990), using the titles of computer science research papers. In the example, the context-vector representations of the terms *user* and *human* have negative correlations, yet their distributional representations have high correlation, which is appropriate since these terms have roughly the same meaning in this dataset.

Word vectors and neural word embeddings

Discriminatively-trained word embeddings very hot area in NLP. The idea is to replace factorization approaches with discriminative training, where the task may be to predict the word given the context, or the context given the word.

Suppose we have the word w and the context c, and we define

$$u_\theta(w, c) = \exp\left(\boldsymbol{a}_w^\top \boldsymbol{b}_c\right) \tag{15.14}$$

$$\tag{15.15}$$

with $\boldsymbol{a}_w \in \mathbb{R}^K$ and $\boldsymbol{b}_c \in \mathbb{R}^K$. The vector $\boldsymbol{a}_w$ is then an **embedding** of the word w, representing its properties. We are usually less interested in the context vector $\boldsymbol{b}$; the context can include surrounding words, and the vector $\boldsymbol{b}_c$ is often formed as a sum of context embeddings for each word in a window around the current word. Mikolov et al. (2013a) draw the size of this context as a random number r.

The popular `word2vec` software[3] uses these ideas in two different types of models:

Skipgram model In the skip-gram model (Mikolov et al., 2013a), we try to maximize the log-probability of the context,

$$J = \frac{1}{M} \sum_m \sum_{-c \leq j \leq c, j \neq 0} \log \mathrm{p}(w_{m+j} \mid w_m) \tag{15.16}$$

$$\mathrm{p}(w_{m+j} \mid w_m) = \frac{u_\theta(w_{m+j}, w_m)}{\sum_{w'} u_\theta(w', w_m)} \tag{15.17}$$

$$= \frac{u_\theta(w_{m+j}, w_m)}{Z(w_m)} \tag{15.18}$$

This model is considered to be slower to train, but better for rare words.

CBOW The continuous bag-of-words (CBOW) (Mikolov et al., 2013b,c) is more like a language model, since we predict the probability of words given context.

[3] https://code.google.com/p/word2vec/

$$J =\frac{1}{M}\sum_m \log \mathrm{p}(w_m \mid c) \tag{15.19}$$

$$=\frac{1}{M}\sum_m \log u_\theta(w_m, c) - \log Z(c) \tag{15.20}$$

$$u_\theta(w_m, c) =\exp\left(\sum_{-c\leq j\leq c, j\neq 0} \boldsymbol{a}_{w_m}^\top \boldsymbol{b}_{w_{m+j}}\right) \tag{15.21}$$

The CBOW model is faster to train (Mikolov et al., 2013a). One efficiency improvement is build a Huffman tree over the vocabulary, so that we can compute a hierarchical version of the softmax function with time complexity $\mathcal{O}(\log V)$ rather than $\mathcal{O}(V)$. Mikolov et al. (2013a) report two-fold speedups with this approach.

The recurrent neural network language model (section 5.4) is still another way to compute word representations. In this model, the context is summarized by a recurrently-updated state vector $\boldsymbol{c}_m = f(\boldsymbol{\Theta}\boldsymbol{c}_{m-1} + \mathbf{U}\boldsymbol{x}_m)$, where $\Theta \in \mathbb{R}^{K\times K}$ defines a the recurrent dynamics, $\mathbf{U} \in \mathbb{R}^{K\times V}$ defines "input embeddings" for each word, and $f(\cdot)$ is a non-linear function such as $\tanh$ or sigmoid. The word distribution is then,

$$P(W_{m+1} = i \mid \boldsymbol{c}_m) = \frac{\exp\left(\boldsymbol{c}_m^\top \boldsymbol{v}_i\right)}{\sum_{i'} \exp\left(\boldsymbol{c}_m^\top \boldsymbol{v}_{i'}\right)}, \tag{15.22}$$

where $\boldsymbol{v}_i$ is the "output embedding" of word i.

Estimating word embeddings*

Training word embedding models can be challenging, because they require probabilities that need to be normalized over the entire vocabulary. This implies a training time complexity of $\mathcal{O}(VK)$ for each instance. Since these models are often trained on hundreds of billions of words, with $V \approx 10^6$ and $K \approx 10^3$, this cost is too high. Estimation techniques eliminate the factor V by making approximations.

One such approximation is negative sampling, which is a heuristic variant of noise-contrastive estimation (Gutmann and Hyvärinen, 2012).

We introduce an auxiliary variable D, where

$$D = \begin{cases} 1, & w \text{ is drawn from the empirical distribution } \hat{\mathrm{p}}(w \mid c) \\ 0, & w \text{ is drawn from the noise distribution } q(w) \end{cases} \tag{15.23}$$

Now we will optimize the objective

$$\sum_{(w,c)\in\mathcal{D}} \log P(D=1 \mid c,w) + \sum_{i=1,w'\sim q}^{k} \log P(D=0, \mid c,w'), \tag{15.24}$$

setting

$$P(D=1 \mid c,w) = \frac{u_\theta(w,c)}{u_\theta(w,c)+k\times q(w)} \tag{15.25}$$

$$P(D=0 \mid c,w) = 1-P(D=1 \mid c,w) \tag{15.26}$$

$$= \frac{k\times q(w)}{u_\theta(w,c)+k\times q(w)}, \tag{15.27}$$

where k is the number of noise samples. Note that we have dropped the normalization term $\sum_{w'} u_\theta(w',c)$. Gutmann and Hyvärinen (2012) show that it is possible to treat the normalization term as an additional parameter z_c, which can be directly estimated (see also Vaswani et al., 2013). Andreas and Klein (2015) go one step further, setting $z_c = 1$, in what has been called a "self-normalizing" probability distribution. This might be trouble if we were trying to directly maximize $\log \mathrm{p}(w \mid c)$, but this is where the auxiliary variable formulation helps us out: if we set θ such that $\sum_{w'} u_\theta(w' \mid c) \gg 1$, we will get a very low probability for $P(D=0)$.[todo: needs a little more explanation]

We can further simplify by setting $k=1$ and $q(w)$ to a uniform distribution, arriving at

$$P(D=1 \mid c,w) = \frac{u_\theta(w,c)}{u_\theta(w,c)+1} \tag{15.28}$$

$$P(D=0 \mid c,w) = \frac{1}{u_\theta(w,c)+1} \tag{15.29}$$

The derivative with respect to a is obtained from the objective

$$L =\sum_m \log \mathrm{p}(D=1 \mid c_m, w_m) + \log \mathrm{p}(D=0 \mid c, w') \tag{15.30}$$

$$=\sum_m \log u_\theta(w_m, c_m) - \log(1+u_\theta(w_m,c_m)) - \log(1+u_\theta(w',c_m)) \tag{15.31}$$

$$\frac{\partial L}{\partial \boldsymbol{a}_i} = \sum_{m:w_m=i} \boldsymbol{b}_{c_m} - \frac{1}{1+u_\theta(w_m,c_m)}\frac{\partial u_\theta(i,c_m)}{\partial \boldsymbol{a}_i} + \sum_m \frac{q(i)}{1+u_\theta(i,c_m)}\frac{\partial u_\theta(i,c_m)}{\partial \boldsymbol{a}_i} \tag{15.32}$$

$$= \sum_{m:w_m=i} \boldsymbol{b}_{c_m} - P(D=1 \mid w_m=i, c_m)\boldsymbol{b}_{c_m} - \sum_m q(i)P(D=0 \mid i, c_m)\boldsymbol{b}_{c_m} \tag{15.33}$$

$$=\sum_m (\delta(w_m=i)-q(i))P(D=0 \mid w_m=i, c_m)\boldsymbol{b}_{c_m}. \tag{15.34}$$

The gradient with respect to $\boldsymbol{b}$ is similar. In practice, we simply sample w' at each instance and compute the update with respect to $\boldsymbol{a}_{w_m}$ and $\boldsymbol{a}_{w'}$. In practice, AdaGrad performs well for this optimization.

Connection to matrix factorization*

Recent work has drawn connections between this procedure for training the skip-gram model and weighted matrix factorization approaches (Pennington et al., 2014; Levy and Goldberg, 2014). For example, Levy and Goldberg (2014) show that skip-gram with negative sampling is equivalent to factoring a matrix X, where

$$X_{i,j} =PMI(W=i, C=j) - \log k, \tag{15.35}$$

where k is a constant offset equal to the number of negative samples drawn in Equation 15.24, and PMI is the **pointwise mutual information** of the events of the word $W=i$ and the context $C=j$,

$$PMI(W=i,C=j) =\log \frac{P(W=i,C=j)}{P(W=i)P(C=j)} \tag{15.36}$$

$$=\log \frac{n(W=i,C=j)}{M}\frac{M}{n(W=i)}\frac{M}{n(C=j)} \tag{15.37}$$

$$=\log \frac{n(W=i,C=j)}{n(W=i)}\frac{M}{n(C=j)}. \tag{15.38}$$

Word embeddings can be obtained by solving the truncated singular value decomposition $\mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^\top = \mathbf{X}$, setting the embedding of word i to $\mathbf{u}_i\sqrt(\Sigma_{i,i})$.

This connection suggests that the differences between recent work on neural word embeddings and much older work on Latent Semantic Analysis may be smaller than they initially seemed! Online learning approaches such as negative sampling stream over data, and require hyperparameter tuning to set the appropriate learning rate. On the other hand, PMI is undefined for word-context pairs that are unobserved (due to the logarithm of zero), requiring a heuristic solution such as positive PMI, $PPMI(i,j) = \max(0, PMI(i,j))$, or shifted positive PPMI $SPPMI_k(i,j) = \max(0, PMI(i,j) - \log k)$. Levy and Goldberg (2014) find that singular value decomposition on shifted positive PMI does better than skipgram negative sampling on some lexical semantic tasks, but worse on others.

Chapter 16

Discourse

16.1 Discourse relations in the Penn Discourse Treebank

- introduce discourse relations
- PDTB annotation framework in D-LTAG
- PDTB parsing

16.2 Rhetorical Structure Theory

- Higher-level discourse structure
- Shift-reduce parsing
- Applications to summarization

16.3 Centering

- Pronouns, forms of reference
- Smooth/rough transitions
- Entity grid implementation

16.4 Lexical cohesion and text segmentation

16.5 Dialogue

Minimal discussion of speech acts etc.

Chapter 17

Anaphora and Coreference Resolution

Pronouns are one of the most noticeable forms of linguistic ambiguity. A Google search for “ambiguous pronoun” reveals dozens of pages warning you to avoid ambiguity. But as we have seen, people resolve all but the most egregious linguistic ambiguities intuitively, below the level of conscious thought.

Moreover, reference ambiguities need not apply only to pronouns. Consider the following text:

(17.1) *Apple Inc Chief Executive Tim Cook has jetted into China for talks with government officials as* ***he***$_1$ *seeks to clear up a pile of problems in* ***[[the firm’s]***$_2$ ***biggest growth market]***$_3$*.*

Some questions:

- Who is referred to by *he*$_1$?
- What entity is referred to by *the firm*$_2$?
- What is Apple’s biggest growth market?

You probably answered these questions by making some commonsense assumptions. Tim Cook is the only individual mentioned, so the personal pronoun *he* probably refers to him; Apple is the only firm mentioned, so *the firm* probably refers to it; a CEO wouldn’t fly to China in order to resolve problems in some other growth market, so *the firm’s biggest growth market* probably refers to China.[1][todo:

[1] These judgments are formalized in Grice’s Maxim of Quantity: make your contribution as informative as required, but not more so.

this is not a great example; try to find one with ambiguity that requires more than Grice to resolve.]

We can use this example to introduce some terminology:

Referring expressions include *he*, *Tim Cook*, *the firm*, *the firm's biggest growth market*. These are surface strings in the text.

Referents include TIM-COOK, APPLE, CHINA; in formal semantics, these may be viewed as objects in a model, such as a database of entities. But referents need not always be entities, as we will see.

Coreference is a property of pairs of referring expressions, which holds when they refer to the same underlying entity.

Anaphora are referring expressions whose meaning depends on another expression in context, which occurs earlier in the document or talk. **Cataphora** refer to expressions that occur later in the document, like *After she won the lottery, Susan quit her job.* **Exophora** refer to entities not defined in the linguistic context.

17.1 Forms of referring expressions

There are many possibilities for describing a referent.

Indefinite NPs *a visit*, *two stores*

Definite NPs *the capital*, *his first trip*

Pronouns *he, it*

Demonstratives *this chainsaw*, *that abandoned mall*

Names *Tim Cook*, *China*

Language users make decisions about which type of referring expression to use, and this is an important challenge for automatic text generation. You can't say,

(17.2) *Rob Ford apologized for "a lot of stupid things" but Rob Ford only acknowledged a video showing Rob Ford smoking what appears to be crack cocaine to demand police release it.*

The **specific** referring expression within a type is determined by syntax and semantic constraints, but the **type** of referring expression (pronoun, name, etc) is largely determined by comprehensibility for the listener. Grice's **Maxim of Quantity** requires that speakers be as informative as necessary, but not more so. It is debatable whether this maxim is precise enough to be formalized computationally, but it loosely suggests that speakers should not use a full name (e.g., *Rob Ford*) when a pronoun will do.

One theory about the relationship between discourse structure and forms of referring expressions is the Givenness Hierarchy (Gundel et al., 1993). This theory is based on the **status** of the referent with respect to both the discourse and the hearer.

Type identifiable (you know what dogs are): indefinite

(17.3) *I couldn't sleep, **a dog** kept me awake.*

Referential (some particular dog): indefinite *this*

(17.4) *I couldn't sleep, **this dog** kept me awake.*

Uniquely identifiable definite

(17.5) *I couldn't sleep, **the neighbor's dog** kept me awake.*

Familiar distal demonstrative

(17.6) ***That dog** next door kept me awake all night.*

Activated demonstrative

(17.7) *My neighbor bought a new dog, and **that dog** kept me awake last night.*

In focus pronoun

(17.8) *Her dog barks constantly. **It** kept me awake all night.*

The location of an entity in the givenness hierarchy depends (in part) on the discourse. Compare the following examples:

(17.9) *You look tired, did a dog keep you awake?*

(17.10) *We bought a dog. It keeps me up all night.*

Referents which were recently accessed acquire *salience*, and are more likely to be near the top of the givenness hierarchy (more on salience later). However, background knowledge also plays an important role: for example, if a pair of speakers lives with a (single) dog, it is always at least uniquely identifiable. Entities may also be **inferrable** from the discourse:

(17.11) *She just bought a new bike.*
__The wheels__ are made of bamboo fiber.

Centering theory*

Centering theory (Grosz et al., 1995) formalizes the notion of salience, by incorporating the syntactic role of each referring expression.

At each utterance U_n, we have:

- A backward-looking center $C_b(U_n)$:
  the entity currently **in focus** after U_n.
- A forward-looking center $C_f(U_n)$:
  an ordered list of candidates for $C_b(U_{n+1})$.
- The top choice in $C_f(U_n)$ is $C_p(U_{n+1})$

How do we order the candidates from $C_b(U_{n+1})$ to the forward-looking center? By syntax:

1. Subject
   ***Abigail** saw an elephant.*
2. Existential predicate nominal
   *There is **an elephant** in the room.*
3. Direct object
   *Abigail gave **a snack** to the elephant.*
4. Indirect object or oblique
   *Abigail gave a snack to **the elephant**.*
5. demarcated adverbial prepositional phrase
   *Inside **the zoo**, the elephant is king.*

Rule: If any element of $C_f(U_n)$ is realized by a pronoun in U_{n+1}, then $C_b(U_{n+1})$ must also be realized as a pronoun.

- Generate possible C_b and C_f for each set of reference assignments
- Filter by constraints: syntax, semantics, and centering rules
- Rank by transition orderings: continue, retain, smooth-shift, rough-shift

	$C_b(U_{n+1}) = C_b(U_n)$ or $C_b(U_n) = \varnothing$	$C_b(U_{n+1}) \neq C_b(U_n)$
$C_b(U_{n+1}) = C_p(U_{n+1})$	Continue	Smooth-shift
$C_b(U_{n+1}) \neq C_p(U_{n+1})$	Retain	Rough-shift

In a coherent discourse, we select transitions according to the following preferences: continue, retain, smooth-shift, rough-shift

Here's an example of how to use centering to resolve pronouns.

U_n	$C_f(U_n)$	$C_p(U_n)$	$C_b(U_n)$	transition
John saw a beautiful Masi at the bike shop	John, Masi, bike shop	John	$\varnothing$	
He showed it to Bob	John, Masi, Bob	John	John	Continue
He showed it to Bob	John, bike shop, Bob	John	John	Continue
He bought it	John, Masi or bike shop	John	John	Continue
He bought it	Bob, Masi or bike shop	Bob	Bob	Smooth-shift

- Centering theory tells us that we prefer *John* over *Bob* as the referent for *he* in U_3, because this would be a continue transition rather than a smooth-shift.
- Centering doesn't really give us a rule for choosing *Masi* over *bike shop* in U_2, because neither is $C_b(U_2)$. We might apply the grammatical role hierarchy since there is no other basis for this decision.

17.2 Pronouns and reference

Are all referents entities? No.

(17.12) *They told me that I was too ugly, but I didn't believe* ***it****.*

(17.13) *Alice saw Bob get angry, and I saw* ***it*** *too.*

(17.14) *They told me that I was too ugly, but* ***that*** *was a lie.*

(17.15) *Jess said she worked in security.*
I suppose ***that****'s one way to put it.*

Are all pronouns referential? Also no. **Cataphora** are references to entities which are evoked after the reference.

(17.16) *When she learned what had happened, Alice took the first bus out of town.*

Some pronouns have **generic** referents.

(17.17) *A good father takes care of __his__ kids.*

(17.18) *I want to buy a Porsche, __they__ are so fast.*

(17.19) *On the moon, __you__ have to carry __your__ own oxygen.*

(17.20) *No wise man who owns a donkey beats it.* Grosz et al. (2014)

Some pronouns don't refer to anything at all.

(17.21) *__It__'s raining.*

(17.22) *__It__'s crazy out there.*

(17.23) *__It__'s money that she's really after.*

(17.24) *__It__ sucks that we have to work so hard.*

In the first two examples above, *it* is **pleonastic**; the third and fourth examples are **cleft** and **extraposition**. How can we automatically distinguish these usages of *it* from referential pronouns? Bergsma et al. (2008) propose a substitutability text. Consider the the difference between the following two examples:

(17.25) *You can make it in advance.*

(17.26) *You can make it in showbiz.*

In the second example, the pronoun **it** is non-referential. One way to see this is by substituting another pronoun, like **them**, into these examples:

(17.27) *You can make them in advance.*

(17.28) ?*You can make them in showbiz.*

The questionable grammaticality of the second example suggests that **it** cannot be referential. Bergsma et al. (2008) operationalize this idea by examining all 5-gram context patterns around instances of the word **it**. Given the example,

(17.29) *... said here Thursday that __it__ is unnecessary to continue*

they construct the following 5-grams:

said	here	Thursday	that	*				
	here	Thursday	that	*	is			
		Thursday	that	*	is	unnecessary		
			that	*	is	unnecessary	to	
				*	is	unnecessary	to	continue

For each of the five patterns around a word, they compute the corpus counts of five **pattern fillers**: *it/its*; *they/them/their*; other pronouns *she/her/...*; rare words (almost always nouns); all other tokens (usually nouns). These 25 counts are converted into a feature vector. They can then train a classifier on these features, using labels of whether specific instances of *it* are referential, achieving a cross-validation accuracy of 86%.

17.3 Resolving ambiguous references

Anaphora resolution is the task of resolving anaphoric references, mainly pronouns like *it*, *this*, and *her*. **Coreference resolution** is a broader task, adding two additional phenomena:

- **Names**: *Barack Obama*, *Obama*, *President Obama*, *Barry O*, *Nobama*
- **Nominals**: *the 44th president*, *the former senator from Illinois*, *our first African-American president*

With these tasks in mind, let's go back to our example:

(17.30) Apple Inc Chief Executive Tim Cook has jetted into China for talks with government officials as **he** seeks to clear up a pile of problems in the firm's biggest growth market, from **its** contested iPad trademark to treatment of local labor. Cook is on **his** first trip to the country...

We have the following anaphoric resolution challenges:

- *he* $\overset{?}{=}$ *Apple Inc*, *Tim Cook*, *China*, *talks*, *government officials*, *government*, ...
- *its* $\overset{?}{=}$ *the firm's biggest growth market*, *the firm*, *problems*, *a pile of problems*, ...
- *his* $\overset{?}{=}$ *Cook*, *local labor*, *its contested iPad trademark*, *iPad*, ...

How can we resolve these references? Anaphora resolution is typically handled by a combination of hard constraints and soft preferences, reflecting different classes of linguistic phenomena.

Constraints

Semantic constraints include morphologically marked information such as number, person, gender, and animacy.

(17.31) Tim Cook has jetted in for talks with officials as **he** seeks to clear up a pile of problems...

We can identify the following features of the pronoun and possible referents:

- Number(*he*) = singular
- Number(*officials*) = plural
- Number(*Tim Cook*) = singular

Since there are no other possible referents, *he* almost certainly refers back to *Tim Cook*. This is occasionally tricky in the case of mass nouns, such as

(17.32) *New York has won the superbowl.*
They are the world champions.

Other features include person, gender, and animacy, as in the following examples:

(17.33) *We$_1$ *told* them$_1$ *not to go*.

(17.34) *Sally met my brother. He charmed her.*

(17.35) *Sally met my brother. She charmed him.*

(17.36) *Putin brought a bottle of vodka. It was from Russia.*

Aside from semantics, there are general constraints on reference within sentences, which seem to generalize well across languages. To understand these constraints, we need to introduce some linguistic terminology:

- x **c-commands** y iff the first branching node above x also dominates y.
- x **binds** y iff x and y are co-indexed and x c-commands y
- if y is not bound, it is **free**

For example, consider the tree in Figure 17.1. In this example, *Mary* c-commands *her*/*herself*, because the first branching node above *Mary* also dominates *her/herself*. However, *her*/*herself* does not c-command *Mary*. Thus, the pronoun *her* **cannot** refer to *Mary*, because pronouns cannot refer to antecedents that c-command them. On the other hand, *herself* **must** refer to *Mary*.

Now consider the example, shown in Figure 17.2. Here, *Mary* does **not** c-commands *her*, but *Mary's mom* c-commands *her*. Thus, *her* **can** refer to *Mary* — and we cannot use reflexive *herself* in this context, unless we are talking about Mary's mom. But note that *her* does not have to refer to *Mary* (unlike the reflexive the pronoun).

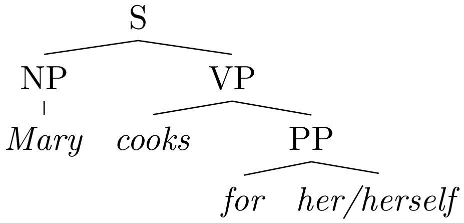


Figure 17.1: *Mary* c-commands *her/herself*

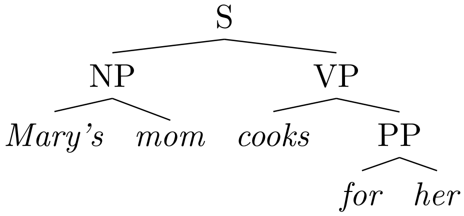


Figure 17.2: *Mary* does not c-command *her*, but *Mary's mom* does.

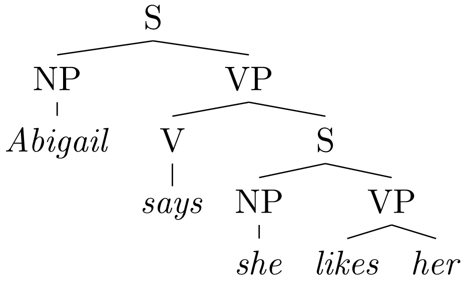


Figure 17.3: A more complex example

A more complex example is shown in Figure 17.3. This indicates how the constraints defined here have a limited domain. The pronoun *she* can refer to *Abigail*, because *Abigail* is outside the domain of *she*. Similarly, *her* can also refer to *Abigail*. But *she* and *her* cannot be coreferent.

Preferences

Putting it together

Three **types** of evidence:

- Semantic constraints
- Syntactic constraints
- Discourse/salience preferences

How do we combine them?

- **Hobbs**: Tree search + constraints

  Walk back through the tree in a deterministic order, select the first referent that satisfies the constraints.

- **Centering**: ordered preferences + constraints

  Apply centering theory to recover the references that give the most preferred transition sequence, subject to semantic constraints.

- **Lappin and Lease**: numerical preferences + constraints

  Basically a hand-tuned linear classifier.

  - -100 for each intervening sentence
  - +80 for subject position
  - +70 for existential emphasis, e.g. *there was a woman who...*
  - +50 for accusative emphasis
  - ...

- Ge, Hale, and Charniak (1999): statistical combination of four probabilities

  - probability of the "Hobbs distance" between pronoun and antecedent
  - probability of the pronoun given the antecedent
    (this considers gender and animacy)
  - how well the proposed antecedent fills the pronoun's slot in the sentence
  - frequency of the proposed referent

- Raghunathan et al. (2010) describe a "multipass sieve" for coreference resolution, which applies a series of progressively relaxed matching rules.

17.4 Coreference resolution

This is a generalization of the anaphora resolution task to cover proper nouns and nominals.

- See the slides for an example.
- The coreference task comes from the information extraction community.
- Candidate spans of text for coreference are called **markables**
- In the harder versions of the coreference task, you have to identify the markables as well as their reference chains.

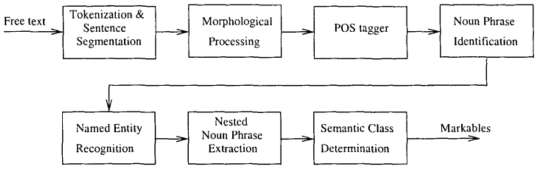
Coreference combines many phenomena: all the ones in anaphora resolution, plus string similarity and knowledge to get nominals.

- *unencrypted Wi-Fi networks* and *networks* have the same head word
- *Dr. King* and *Martin Luther King* can all co-refer
- *Martin Luther King* and *Coretta Scott King* cannot
- **World knowledge**: e.g., *Google* is a *company*,
  companies possess *cars* but *Tuesday* doesn't.

The mention-pair model

One of the earliest end-to-end machine learning systems for coreference is from Soon et al. (2001).

- Identify markables and their features with an NLP pipeline.


Free text
Tokenization & Sentence Segmentation
Morphological Processing
POS tagger
Noun Phrase Identification
Named Entity Recognition
Nested Noun Phrase Extraction
Semantic Class Determination
Markables

- Train a classifier to predict which pairs of markables corefer. This is the **mention-pair** model.
  - For each markable, go backwards until the classifier selects an antecedent or you reach the beginning of the document.

  - No structured prediction here; each classification decision is made independently.

Learning is performed on mention pairs.

- Given the labeled chain A1-A2-A3-A4, the adjacent pairs A1-A2, A2-A3, A3-A4 are treated as positive examples.
- Negative examples are generated from NPs that occur between the adjacent pairs.
  - Suppose markables A,B,B1 appear between A1 and A2.
  - Then the negative examples are: A-A2, B-A2, B1-A2.

There are fundamental problems with mention-pair approaches.

- They fail to aggregate information across the chain.
- Must reason about transitivity to avoid incoherent chains.
- *Michelle Obama* ← *Obama* ← *Mr. Obama*

Entity-based coreference

Alternatively, we can try to learn at the entity level, using features of the entities themselves

- Number of entities detected so far
- Mention to entity ratio
- Entity to word ratio
- Number of intervening mentions between mention and linked entity
- ...

Can incorporate these by scoring entire clusterings, $\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{y})$.
But how to train such a model?

One approach is an incremental perceptron. This is like a structured perceptron, but you incrementally build the structure, and you update as soon as you make a mistake.

Bell Tree, Beam Search, and Max-link Coreference The Bell Tree can represent the coreference structure. See slides.

Markov Random Field with Transitive Closure see slides

Summing over antecedent structures Durrett and Klein (2013) propose summing over reference assignments within a clustering. Let the gold standard clustering be written C^*, with C^*_k representing the cluster for document k, and $\mathcal{A}(C^*_k)$ representing the set of possible antecedents structures. Then we treat the specific antecedent structure as a latent variable, and sum over it, obtaining the regularized objective,

$$\ell(\boldsymbol{\theta}) = \sum_k^N \log \left(\sum_{a \in \mathcal{A}(C^*_k)} \mathrm{p}(a \mid x_k) \right) + \lambda ||\boldsymbol{\theta}|| \tag{17.1}$$

$$\mathrm{p}(a \mid x_k) \propto \exp \left(\sum_i \boldsymbol{\theta}^\top \boldsymbol{f}(i, a_i, \boldsymbol{x}) \right). \tag{17.2}$$

Durrett and Klein (2013) augment this basic model by defining a real-valued loss function, and incorporate it into the objective. [todo: say a little more] They then show that this basic framework supports a number of expressive features, which give good performance compared to prior work.

Durrett and Klein (2013) also note that the most challenging cases by far are nominals that are anaphoric, but in which the head word has not appeared before. For example,

(17.37) *Tim Cook visited China yesterday.*
The Apple CEO said that international cooperation was a high priority for his company.

Here *CEO* is the head of the nominal NP, *the apple CEO*, which refers to *Tim Cook*. Clearly, this case is hard to resolve without external world knowledge. Durrett and Klein (2013) call this an "uphill battle", in contrast to the "easy victories" attainable in the case of pronoun resolution. Haghighi and Klein (2009) mine Wikipedia data to try to learn enough world knowledge to handle these cases.

17.5 Coreference evaluation

17.6 Multidocument coreference resolution

Part IV

Applications

Chapter 18

Information extraction

A fundamental challenge for artificial intelligence (AI) is **knowledge acquisition**: how to give computers enough knowledge so as to make their inferential capabilities useful (**?**). From an AI perspective, one of the major motivations for natural language processing is to provide a solution to this problem — acquiring knowledge in the way that people often do, by reading. This problem is sometimes called **information extraction**; in contrast to **information retrieval**, where the goal is to retrieve informative documents for a human reader, the goal of information extraction is to synthesize these documents into structured knowledge representations, such as database entries.

This chapter distinguishes information extraction from **question answering**, where the goal is to provide natural language answers to natural language questions. The tasks are closely related: a question answering system might proceed by first parsing the question (determining what information is required), then identifying relevant records in the knowledge base, and then crafting a natural language response. In many scenarios — such as the IBM question answering system "Watson" — the required knowledge base is too large to create by hand, so it must be created by information extraction techniques, similar to those discussed here.

A large part of information extraction can be unified in terms of **entities**, **relations**, and **events**. Entities are uniquely specified objects in the world, such as people, places, organizations, and times. Relations link pairs of entities, as in $\textsf{sibling}(\textsc{Luke}, \textsc{Leia})$. We can think of each relation type as defining a table, in which each row contains two entities. Events link arbitrary numbers of arguments, as in the following example:

$$\begin{aligned}\textsf{battle} : \langle \textsf{location} &: \textsc{Atlanta},\\ \textsf{date} &: 1864,\\ \textsf{victor} &: \textsc{United States Army},\\ \textsf{defeated} &: \textsc{Confederate Army}\rangle.\end{aligned}$$

We can think of each event type as defining a table, in which the rows define various "slots" pertaining to the event. The task of **knowledge base population** is closely related to information extraction, and the goal is to fill in relevant slots in just such a table.

The attentive reader will notice a close kinship between information extraction, as defined here, and the task of shallow semantic parsing defined in chapter 14. For example, in semantic role labeling, the goal was to identify predicates and their arguments; we may think of predicates as corresponding to events, and the arguments as defining slots in the event representation. The key difference is that semantic role labeling and related tasks require correctly analyzing each sentence — a goal sometimes described as **micro-reading**. In information extraction, we need only correctly identify the relations and events that are referred to in a corpus. Many relations and events may be mentioned multiple times, but in information extraction and knowledge base population, we need only identify them once — thus the goal here is sometimes described as **macro-reading**. While macro-reading is a more forgiving task than micro-reading, it requires reasoning over an entire corpus, posing additional problems of computational tractability. It may also be necessary to provide **information provenance** [todo: good term?], linking the extracted knowledge back to the original source or sources.

18.1 Entities

The starting point for information extraction is to identify mentions of entities in text. For example, consider the following text.

(18.1) *The United States Army captured a hill overlooking Atlanta on May 14, 1864.*

Given this text, we have two goals:

1. **Identify** the spans *United States Army*, *Atlanta*, and *May 14, 1864* as entity mentions. We may also want to recognize the **named entity types**: organization, location, and date. This task is known as **named entity recognition**.

The	U.S.	Army	captured	Atlanta	on	May	14	,	1864	.
B-ORG	I-ORG	I-ORG	O	B-LOC	O	B-DATE	I-DATE	I-DATE	I-DATE	O

Table 18.1: BIO notation for named entity recognition

2. **Link** these spans to known entities in a knowledge base, U.S. ARMY, ATLANTA, and MAY 14, 1864. This task is known as **entity linking**.

Named entity recognition (NER)

A standard approach is to tagging named entity spans is to use discriminative sequence labeling methods such as conditional random fields and structured perceptrons. As described in chapter 6, these methods use the Viterbi algorithm to search over all possible label sequences, while scoring each sequence using a feature function that decomposes across adjacent tags. Named entity recognition is formulated as a tagging problem by assinging each word token to a tag from a tagset. However, there is a major difference from part-of-speech tagging: in NER we need to recover **spans** of tokens, such as *The United States Army*. To do this, the tagset must distinguish tokens that are at the **b**eginning of a span from tokens that are **i**nside a span.

BIO notation This is accomplished by the "BIO notation", shown in Table 18.1. Each token at the beginning of a name span is labeled with a B- prefix; each token within a name span is labeled with an I- prefix. Tokens that are not parts of name spans are labeled as O. From this representation, it is unambiguous to recover the entity name spans within a labeled text. Another advantage is from the perspective of learning: tokens at the beginning of name spans may have different properties than tokens within the name, and the learner can exploit this. This insight can be taken even further, with special labels for the **l**ast tokens of a name span, and for **u**nique tokens in name spans, such as *Atlanta* in the example in Table 18.1. This is called BILOU notation, and has been shown to yield improvements in supervised named entity recognition Ratinov and Roth (2009).[todo: check this cite]

Entity types The number of possible entity types depends on the labeled data. An early dataset was released as part of a shared task in the Conference on Natural Language Learning (CoNLL), containing entity types LOC (location), ORG (organization), and PER (person). Later work has distinguished additional entity types, such as dates, [todo: etc]. [todo: find cites] Special purpose corpora have been built for domains such as biomedical text, where entities include protein types [todo: etc].

Features The use of Viterbi decoding restricts the feature function $\boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y})$ to $\sum_m \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m)$, so that each feature can consider only local adjacent tags. Typical features include tag transitions, word features for w_m and its neighbors, character-level features for prefixes and suffixes, and "word shape" features to capture capitalization. As an example, base features for the word *Army* in the example in Table 18.1 include:

$$\begin{aligned}
&\langle \textsc{curr-word}\text{:}\textit{Army}, \\
&\textsc{prev-word}\text{:}\textit{U.S.}, \\
&\textsc{next-word}\text{:}\textit{captured}, \\
&\textsc{prefix-1}\text{:}\textit{A-}, \\
&\textsc{prefix-2}\text{:}\textit{Ar-}, \\
&\textsc{suffix-1}\text{:}\textit{-y}, \\
&\textsc{suffix-2}\text{:}\textit{-my}, \\
&\textsc{shape}\text{:}\textit{Xxxx}\rangle
\end{aligned}$$

Another source of features is to use **gazzeteers**: lists of known entity names. For example, it is possible to obtain from the U.S. Social Security Administration a list of [todo: hundreds of thousands] of frequently used American names — more than could be observed in any reasonable annotated corpus. Tokens or spans that match an entry in a gazetteer can receive special features; this provides a way to incorporate hand-crafted resources such as name lists in a learning-driven framework.

Features in recent state-of-the-art systems are summarized in papers by **?** and Ratinov and Roth (2009).

Alternative modeling frameworks*

Apart from sequence labeling, there are other formulations for named entity recognition, which are arguably better customized for the task.

18.2 Relations

Knowledge-base population

Distant supervision

18.3 Events and processes

18.4 Facts, beliefs, and hypotheticals

Chapter 19

Machine translation

Machine translation (MT) is one of the "holy grail" problems in natural language processing. Solving it would be a major advance in facilitating communication between people all over the world, and so it has received a lot of attention and funding since the early 1950s. However, it has proved incredibly challenging, and while there has been substantial progress towards usable MT systems — especially for so-called "high resource" languages like English and French — we are still far from automatically producing translations that capture the nuance and depth of human language.

19.1 The noisy channel model

Throughout the course, we've been working with the general formulation,

$$\hat{\boldsymbol{y}} = \operatorname{argmax} \boldsymbol{y} \in \mathcal{Y} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{y}) \tag{19.1}$$

Now suppose we make $\mathcal{X}$ equal to the set of all possible sentences in a foreign language, and $\mathcal{Y}$ equal to the set of all possible English sentences. We can thus view translation in the same linear formalism that we've considered all along. Will this work?

There are two major criteria for a translation:

- **Adequacy**: The translation $\hat{\boldsymbol{y}}$ should adequately reflect the linguistic content of $\boldsymbol{x}$. For example, if $\boldsymbol{x} =$ *Vinay le gusta Python*, the *gloss*[1] $\boldsymbol{y} =$ *Vinay it like Python* is considered adequate becomes it contains all the relevant content. The output $\boldsymbol{y} =$ *Vinay debugs memory leaks* will score poorly.

[1] A "gloss" is a word-for-word translation.

- **Fluency**: The translation $\hat{\boldsymbol{y}}$ should read like fluent text in the target language. By this criterion, the gloss $\boldsymbol{y} =$ *Vinay it like Python* will score poorly, and $\boldsymbol{y} =$ *Vinay likes Python* will be preferred.

	Adequate?	Fluent?
Vinay it like Python	yes	no
Vinay debugs memory leaks	no	yes
Vinay likes Python	yes	yes

Table 19.1: Adequacy and fluency for translations of the Spanish *Vinay le gusta Python*

An early insight in machine translation was that the scoring function for a translation can decompose across these criteria:

$$\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{\theta}_t^\top \boldsymbol{f}_t(\boldsymbol{x}, \boldsymbol{y}) + \boldsymbol{\theta}_\ell^\top \boldsymbol{f}_\ell(\boldsymbol{y}) \tag{19.2}$$

The features $\boldsymbol{f}_t$ represent the translation model, which corresponds to the adequacy criterion; the features $\boldsymbol{f}_\ell$ represent the language model, which corresponds to the fluency criterion.

The advantage of this decomposition is that we can estimate $\boldsymbol{\theta}_\ell^\top$ from unlabeled data in the target language. Because unlabeled text data is widely available, in principle we can easily improve the fluency of our translations by estimating very high-order language models from ample unlabeled text. In this case, we can express these features as

$$\boldsymbol{f}_\ell(\boldsymbol{y}) = \bigcup_i \mathbf{1}(\boldsymbol{y}_{i:i+k}) \tag{19.3}$$

$$\theta_\ell(\{w_0, w_1, w_2, \dots w_k\}) = \log \mathrm{p}(w_k \mid w_{k-1}, w_{k-2}, \dots, w_0) \tag{19.4}$$

When estimating these probabilities, we will naturally want to apply all the smoothing tricks that we learned in Chapter 5. Note that we will also have to add padding of K "buffer" words at the beginning and end of the input.

This approach is indeed a component of the current state-of-the-art MT systems, but there is a catch: as the size of the N-gram features increases, the problem of **decoding** — selecting the best scoring translation $\hat{\boldsymbol{y}}$ — becomes exponentially more difficult. We will consider this issue later. For now, just note that this formulation ensures that,

$$\boldsymbol{\theta}_\ell^\top \boldsymbol{f}_\ell(\boldsymbol{y}) = \log \mathrm{p}(\boldsymbol{y}). \tag{19.5}$$

Now let's consider the translation component. If we can set

$$\boldsymbol{\theta}_t^\top \boldsymbol{f}_t(\boldsymbol{y}, \boldsymbol{x}) = \log \mathrm{p}(\boldsymbol{x} \mid \boldsymbol{y}), \tag{19.6}$$

then the sum of these two scores yields,

$$\begin{aligned} \boldsymbol{\theta}_t^\top \boldsymbol{f}_t(\boldsymbol{y}, \boldsymbol{x}) + \boldsymbol{\theta}_\ell^\top \boldsymbol{f}_\ell(\boldsymbol{y}) &= \log \mathrm{p}(\boldsymbol{x} \mid \boldsymbol{y}) + \log \mathrm{p}(\boldsymbol{y}) && (19.7) \\ &= \log \mathrm{p}(\boldsymbol{x}, \boldsymbol{y}). && (19.8) \end{aligned}$$

In other words, we can obtain the translation $\hat{\boldsymbol{y}}$ which has the maximum joint log-likelihood $\log \mathrm{p}(\boldsymbol{y}, \boldsymbol{x})$. We want the translation with the highest conditional probability,

$$\mathrm{argmax}\, \boldsymbol{y} \mathrm{p}(\boldsymbol{y} \mid \boldsymbol{x}) = \mathrm{argmax}\, \boldsymbol{y} \frac{\mathrm{p}(\boldsymbol{y}, \boldsymbol{x})}{\mathrm{p}(\boldsymbol{x})}, \tag{19.9}$$

but since $\boldsymbol{x}$ is given, we can ignore the denominator $\mathrm{p}(\boldsymbol{x})$ and just select the $\boldsymbol{y}$ that maximizes the joint probability.

This approach is called the **noisy channel model**, and was pioneered by researchers who were experts in cryptography. They proposed to view translation as *decoding* the output of a stochastic cipher.

- Imagine that the original text $\boldsymbol{y}$ was written in English, and is modeled as drawn from a source language model $\boldsymbol{y} \sim P_\ell$
- The source was then stochastically encoded, according to the translation model, $\boldsymbol{x} \mid \boldsymbol{y} \sim P_t$.
- If we can estimate the stochastic processes P_ℓ and P_t, we can reverse the cipher and obtain the original text.

19.2 Translation modeling

Language modeling is covered in Chapter 5, so this chapter will mainly focus on the translation model, $\mathrm{p}_t(\boldsymbol{x} \mid \boldsymbol{y})$. To estimate this model, we will need a parallel corpus, which contains sentences in both languages.

- Parallel corpora are often available from national and international governments. **The Hansards corpus** contains aligned English and French sentences from the Canadian parliament. **The EuroParl corpus** contains sentences for 21 languages, aligned with their English translations.

- More recent work has explored the use of web documents (Kilgarriff and Grefenstette, 2003; Resnik and Smith, 2003) and crowdsourcing for MT (Zaidan and Callison-Burch, 2011).

Once a parallel corpus is obtained, we can consider how to characterize the translation model, $\boldsymbol{f}_t$. The sets $\mathcal{X}$ and $\mathcal{Y}$ are far too huge for us to directly estimate the adequacy of every possible translation pair. So we need to decompose this problem into smaller units.

The **Vauquois Pyramid** is a theory of how translation should be modeled. At the lowest level, we translate individual words, but the distance here is far, because languages express ideas differently. If we can move up the triangle to syntactic structure, the distance for translation is reduced; we then need only produce target-language text from the syntactic representation, which can be as simple as reading off a tree. Further up the triangle lies semantics; translating between semantic representations should be easier still, but mapping between semantics and surface text is a difficult, unsolved problem. At the top of the triangle is **interlingua**, a semantic representation that is so generic, it is identical across all human languages. Philosophers may debate whether such a thing as interlingua is really possible (Derrida, 1985), but the idea of linking translation and semantic understanding is viewed by many as a grand challenge for natural language technology.

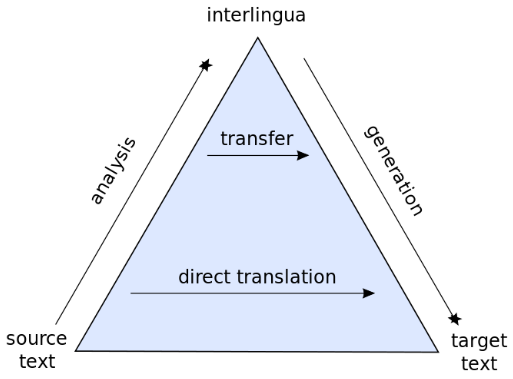


Figure 19.1: The Vauquois Pyramid ("Direct translation and transfer translation pyramind". Licensed under Creative Commons Attribution-Share Alike 3.0 via Wikimedia Commons.)

Returning to earth, the simplest decomposition of the translation model is a word-based translation: each word in the source string should be aligned to a word in the translation. In this approach, we need an alignment $\mathcal{A}(\boldsymbol{x}, \boldsymbol{y})$, which contains a list of pairs of source and target tokens. Making some independence assumptions, we can then define the translation probability as,

$$\begin{aligned}
\mathrm{p}_t(\boldsymbol{x}, \mathcal{A} \mid \boldsymbol{y}) =& \prod_i \mathrm{p}(x_i, a_i \mid y_{a_i}) && (19.10)\\
=& \prod_i \mathrm{p}_a(a_i \mid i, N_x, N_y) \times \mathrm{p}_{x|y}(x_i \mid y_{a_i}) && (19.11)
\end{aligned}$$

Key assumptions:

- The alignment probability decomposes as $\mathrm{p}(\mathcal{A} \mid \boldsymbol{x}, \boldsymbol{y}) = \prod_i \mathrm{p}_a(a_i \mid i, N_x, N_y)$. This means that each alignment decision is independent of the others, and depends only on the index i, and the sentence lengths N_x and N_y.
- The translation probability decomposes as $\mathrm{p}(\boldsymbol{x} \mid \boldsymbol{y}, \mathcal{A}) = \prod_i \mathrm{p}_{x|y}(x_i \mid y_{a_i})$. We are doing word-based translation only, ignoring context. The hope is that the language model will correct any disfluencies that arise from word-to-word translation.

A series of translation models with increasingly weak independence assumptions was produced by researchers at IBM in the 1980s and 1990s, known as IBM Models 1-6(Och and Ney, 2003). IBM model 1 makes the strongest independence assumption:

$$\mathrm{p}_a(a_i \mid i, N_x, N_y) = \frac{1}{N_y} \tag{19.12}$$

In this model every alignment is equally likely! This is almost surely wrong, but it makes learning easy.

Let's consider how to translate with IBM model 1. The key idea is to treat the alignment as a **hidden variable**. If we knew the alignment, we could easily estimate a translation model, and we could find the optimal translation as

$$\begin{aligned}
\hat{\boldsymbol{y}} =& \operatorname{argmax} \boldsymbol{y} \mathrm{p}(\boldsymbol{x}, \boldsymbol{y}) && (19.13)\\
=& \operatorname{argmax} \boldsymbol{y} \sum_{\mathcal{A}} \mathrm{p}(\boldsymbol{x}, \boldsymbol{y}, \mathcal{A}) && (19.14)\\
=& \operatorname{argmax} \boldsymbol{y} \mathrm{p}_\ell(\boldsymbol{y}) \sum_{\mathcal{A}} \mathrm{p}_t(\boldsymbol{x}, \mathcal{A} \mid \boldsymbol{y}) && (19.15)\\
\approx& \operatorname{argmax} \boldsymbol{y} \mathrm{p}_\ell(\boldsymbol{y}) \max_{\mathcal{A}} \mathrm{p}_t(\boldsymbol{x}, \mathcal{A} \mid \boldsymbol{y}) && (19.16)
\end{aligned}$$

Conversely, if we had an accurate translation model, we could estimate beliefs about each alignment decision,

$$q(a_i \mid \boldsymbol{x}, \boldsymbol{y}) \propto \mathrm{p}_a(a_i \mid i, N_x, N_y) \times \mathrm{p}_{x|y}(\boldsymbol{x}_i \mid \boldsymbol{y}_{a_i}). \tag{19.17}$$

We therefore have a classic chicken-and-egg problem, which we can solve using the iterative expectation-maximization (EM) algorihtm.

E-step Update beliefs about word alignment,

$$q_i(a_i) \propto \mathrm{p}_a(a_i \mid i, N_x, N_y)\mathrm{p}_{x|y}(\boldsymbol{x}_i \mid \boldsymbol{y}_{a_i}) \tag{19.18}$$

M-step Update the translation model,

$$\theta_{u \to v} = \log \frac{\sum_i \sum_j q_i(a_i = j)\delta(y_j = u \wedge x_i = v)}{\sum_i \sum_j q_i(a_i = j)\delta(y_j = u)} \tag{19.19}$$

Example for IBM Model 1

Suppose our bitext has two sentence pairs:

(19.1) *The coffee*
Le cafe

(19.2) *My coffee*
Mon cafe

We start with the following translation probabilities:

	le	*mon*	*cafe*
the	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$
my	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$
coffee	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$

In the E-step, we compute alignment probabilities for each sentence.

$$\begin{align}
q_0(0) \propto & \mathrm{p}_a(0) \times \mathrm{p}(\textit{le} \mid \textit{the}) = \frac{1}{2} \times \frac{1}{3} \tag{19.20}\\
q_0(1) \propto & \mathrm{p}_a(1) \times \mathrm{p}(\textit{le} \mid \textit{coffee}) = \frac{1}{2} \times \frac{1}{3} \tag{19.21}\\
q_0(\cdot) = & \left[\frac{1}{2}, \frac{1}{2}\right] \tag{19.22}
\end{align}$$

	le	*mon*	*cafe*
the	$\frac{1}{2}$	0	$\frac{1}{2}$
my	0	$\frac{1}{2}$	$\frac{1}{2}$
coffee	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{2}$

The same logic applies to all the alignment decisions: we begin with $q_i(j) = \frac{1}{N}$ in every case. Now we move to the M-step, where we will plug in these (apparently uninformative) alignment probabilities:

$$\mathrm{p}_{x|y}(\textit{le} \mid \textit{the}) = \frac{\sum_{i,j} q_i(j)\delta(y_i = \textit{le} \wedge x_j = \textit{the})}{\sum_{i,j} q_i(j)\delta(x_j = \textit{the})} = \frac{\frac{1}{2}}{\frac{1}{2} + \frac{1}{2}} = \frac{1}{2} \tag{19.23}$$

$$\mathrm{p}_{x|y}(\textit{cafe} \mid \textit{the}) = \frac{\sum_{i,j} q_i(j)\delta(y_i = \textit{le} \wedge x_j = \textit{the})}{\sum_{i,j} q_i(j)\delta(x_j = \textit{the})} = \frac{\frac{1}{2}}{\frac{1}{2} + \frac{1}{2}} = \frac{1}{2} \tag{19.24}$$

$$\mathrm{p}_{x|y}(\textit{mon} \mid \textit{the}) = \frac{\sum_{i,j} q_i(j)\delta(y_i = \textit{le} \wedge x_j = \textit{the})}{\sum_{i,j} q_i(j)\delta(x_j = \textit{the})} = \frac{0}{\frac{1}{2} + \frac{1}{2}} = 0 \tag{19.25}$$

The math works out similarly for $\mathrm{p}(\cdot \mid \textit{my})$. But the English word *coffee* appears in both sentence pairs, so:

$$\mathrm{p}_{x|y}(\textit{le} \mid \textit{cafe}) = \frac{\frac{1}{2}}{4 \times \frac{1}{2}} = \frac{1}{4} \tag{19.26}$$

$$\mathrm{p}_{x|y}(\textit{coffee} \mid \textit{cafe}) = \frac{2 \times \frac{1}{2}}{4 \times \frac{1}{2}} = \frac{1}{2} \tag{19.27}$$

$$\mathrm{p}_{x|y}(\textit{mon} \mid \textit{cafe}) = \frac{\frac{1}{2}}{4 \times \frac{1}{2}} = \frac{1}{4} \tag{19.28}$$

$$\tag{19.29}$$

To summarize the new translation probabilities:

	le	*mon*	*cafe*
the	$\frac{2}{3}$	0	$\frac{1}{3}$
my	0	$\frac{2}{3}$	$\frac{1}{3}$
coffee	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{2}{3}$

We now go back to the E-step and compute the alignments again.

$$\begin{aligned}
q_0(0) \propto & \mathrm{p}_a(0) \times \mathrm{p}(\textit{le} \mid \textit{the}) = \frac{1}{2} \times \frac{1}{2} && (19.30)\\
q_0(1) \propto & \mathrm{p}_a(1) \times \mathrm{p}(\textit{le} \mid \textit{coffee}) = \frac{1}{2} \times \frac{1}{4} && (19.31)\\
q_0(\cdot) = & \left[\frac{2}{3}, \frac{1}{3}\right] q_1(0) \propto \qquad \mathrm{p}_a(0) \times \mathrm{p}(\textit{le} \mid \textit{coffee}) = \frac{1}{2} \times \frac{1}{4} && (19.32)\\
q_1(1) \propto & \mathrm{p}_a(1) \times \mathrm{p}(\textit{cafe} \mid \textit{coffee}) = \frac{1}{2} \times \frac{1}{2} && (19.33)\\
q_1(\cdot) = & \left[\frac{1}{3}, \frac{2}{3}\right] && (19.34)
\end{aligned}$$

Having learned something about the translation model, the alignments are no longer uniform. The situation for the second sentence is identical, so is not shown here.

If we return to the M-step, we end up with sharper translation probabilities:

$$\begin{aligned}
\mathrm{p}_{x|y}(\textit{le} \mid \textit{the}) = & \frac{\sum_{i,j} q_i(j)\delta(y_i = \textit{le} \wedge x_j = \textit{the})}{\sum_{i,j} q_i(j)\delta(x_j = \textit{the})} = && \frac{\frac{2}{3}}{\frac{2}{3} + \frac{1}{3}} = \frac{2}{3} && (19.35)\\
\mathrm{p}_{x|y}(\textit{cafe} \mid \textit{the}) = & \frac{\sum_{i,j} q_i(j)\delta(y_i = \textit{le} \wedge x_j = \textit{the})}{\sum_{i,j} q_i(j)\delta(x_j = \textit{the})} = && \frac{\frac{1}{3}}{\frac{1}{3} + \frac{2}{3}} = \frac{1}{3} && (19.36)\\
\mathrm{p}_{x|y}(\textit{mon} \mid \textit{the}) = & 0 && && (19.37)\\
\mathrm{p}_{x|y}(\textit{le} \mid \textit{cafe}) = & \frac{\frac{1}{3}}{\frac{1}{3} + \frac{2}{3} + \frac{1}{3} + \frac{2}{3}} = \frac{1}{6} && && (19.38)\\
\mathrm{p}_{x|y}(\textit{coffee} \mid \textit{cafe}) = & \frac{2 \times \frac{2}{3}}{2} = \frac{2}{3} && && (19.39)\\
\mathrm{p}_{x|y}(\textit{mon} \mid \textit{cafe}) = & \frac{\frac{1}{3}}{2} = \frac{1}{6} && && (19.40)
\end{aligned}$$

The process will eventually converge to assign all of the probability mass for each English word to its correct French translation. Note that we have made no assumptions about the word alignments at all! The only information that we have exploited is the co-occurrence of words across sentence pairs. But we can do even better in models that make reasonable assumptions about alignment — for example, that alignments tend to be monotonic ($i > j \rightarrow a_i > a_j$), etc.

Better alignment models IBM Model 2 tries to learn the prior distribution from data,

$$\mathrm{p}_a(a_i; i, N_x, N_y) = \phi_{a_i,i,N_x,N_y} \tag{19.41}$$

$$s.t. \forall i, N_x, N_y, \sum_a \phi_{a,i,N_x,N_y} = 1. \tag{19.42}$$

The solution is the expected relative frequency estimate,

$$\phi_{a,i,N_x,N_y} = \frac{\sum_{\boldsymbol{y},\boldsymbol{x}:\#|\boldsymbol{y}|=N_y,\#|\boldsymbol{x}|=N_x} q_i(a)}{\sum_{\boldsymbol{y},\boldsymbol{x}:\#|\boldsymbol{y}|=N_y,\#|\boldsymbol{x}|=N_x}}, \tag{19.43}$$

where we are summing only over sentence pairs with lengths N_x, N_y.

Adding a parameter for the alignment model makes the overall objective function non-convex (see chapter 4 for a review of convexity). The practical consequence of this is that initialization matters; it's no longer sufficient to just initialize the translation model to uniform probabilities and hope that everything works out. A good solution is to first run IBM Model 1, and then use the resulting translation model as the initialization for IBM Model 2.

IBM model 3 adds a term for the "fertility" of each word — that is, the number of words that typically align to it. For example, some English verbs are translated as multiword phrases:

(19.3) *Mary did not **slap** the green witch.*
*Maria no **daba una bofetada** a la bruja verde.*

By learning these fertility probabilities from data, the alignment model has a better chance of learning the correct translation rules for such multiword phrases. But note that even in the best case, we would have to model the translation of *slap* into *daba una bofetada* as,

$$\mathrm{p}_{x|y,\mathcal{A}}(\textit{daba una bofetada} \mid \textit{slap}) \tag{19.44}$$

$$= \mathrm{p}_{x|y}(\textit{daba} \mid \textit{slap}) \times \mathrm{p}_{x|y}(\textit{una} \mid \textit{slap}) \times \mathrm{p}_{x|y}(\textit{bofetada} \mid \textit{slap}). \tag{19.45}$$

This seems wrong, since the word *una* is just an indefinite article — the Spanish feminine for the English word *a*. We therefore turn to models that go beyond word-based translation.

19.3 Phrase-based translation

The problem identified with the example *daba una bofetada* is an instance of a more general issue: translation is often not a matter of word to word substitutions. Multiword expressions are often not translated literally:

(19.4) *clean up*
faire (make) le (the) menage (home)

Handling this in a word-to-word translation model seems unnecessarily difficult. Furthermore, phrases tend to move together:

(19.5) *i like the food a lot*
la (the) comida (food) me (I) gusta (like) mucho (a lot)

We would therefore have to learn that the alignment decisions for *la* and *comida* should be made jointly.

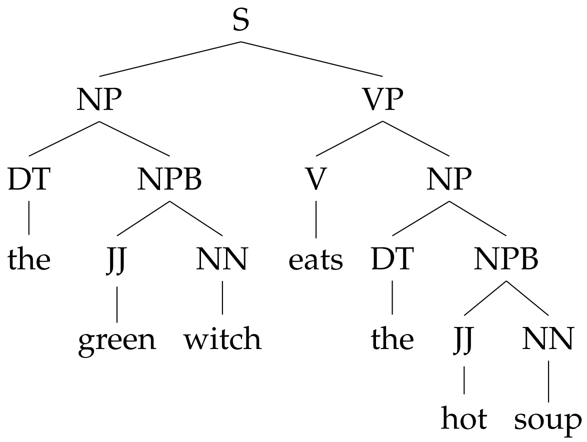
Phrase-based translation generalizes on word-based models by building translation tables and alignments between multiword spans of text. The generalization from word-based translation is surprisingly straightforward: the translation tables can now condition on multi-word units, and can assign probabilities to multiword units; alignments are mappings from spans to spans, $\langle (i, j), (k, \ell) \rangle$, so that

$$\mathrm{p}(\boldsymbol{x} \mid \boldsymbol{y}, \mathcal{A}) = \prod_{\langle (i,j),(k,\ell) \rangle \in \mathcal{A}} \mathrm{p}_{x|y}(\{x_i, x_{i+1}, \ldots, x_j\} \mid \{y_k, y_{k+1}, \ldots, y_\ell\}), \tag{19.46}$$

where we require that the alignment set $\mathcal{A}$ cover both sentences with non-overlapping spans, as shown in **??**. [todo: add figure]

19.4 Syntactic MT

Consider the English sentence, *The green witch eats the hot soup.*



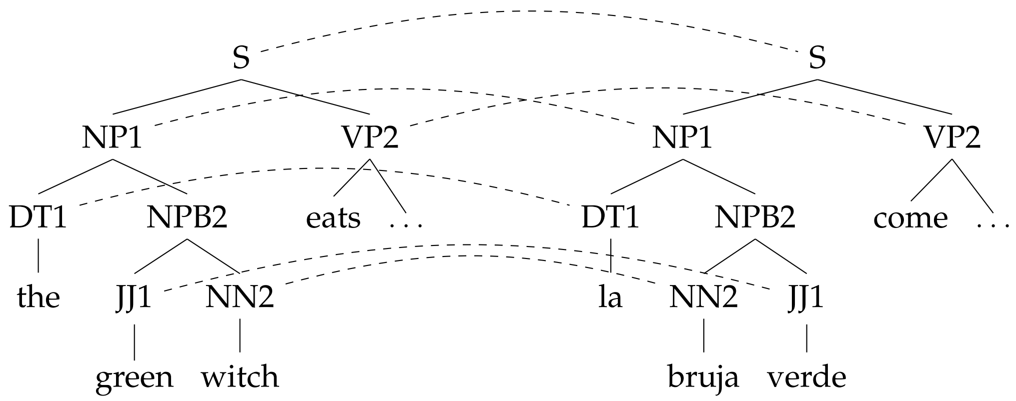
Where NPB is a "bare NP," without the determiner. We might get this non-terminal from binarizing a CFG.

We can view the CFG as a process for **generating** English sentences.

Synchronous CFGs are a generalization of CFGs. They generate text in two different languages simultaneously. Each RHS has two components, one for each language. Subscripts show the mapping between non-terminals in the RHS. For example:

$$\begin{aligned}
S &\to NP_1\, VP_2, & NP_1\, VP_2\\
VP &\to V_1\, NP_2, & V_1\, NP_2\\
NP &\to DT_1\, NPB_2, & DT_1\, NPB_2\\
NPB &\to JJ_1\, NPB_2, & NPB_2\, JJ_1
\end{aligned}$$

The key production is the fourth one, which handles the re-ordering of adjectives and nouns. Let's use this SCFG to generate the English and Spanish versions of this sentence.



- On the slides there is another example, in Japanese. Since Japanese is a SOV language (subject-object-verb), we need a production: $VP \rightarrow V_1\, NP_2, NP_2\, V_1$.
- As with CFGs, we can attach a probability to each production, and compute the joint probability of the derivation and the text as the product of these productions.

Binarization

Let's define a rank-n CFG as a grammar with at most n elements on a right-hand side.

- CFGs can always be binarized.
  - e.g. $NP \rightarrow DT\ [JJ\ NN]$ becomes

$$\begin{aligned} NP &\rightarrow DT\ NPB \\ NPB &\rightarrow JJ\ NN \end{aligned}$$

  - Therefore, the set of languages that can be defined by a 2-CFG is identical to the set that can be defined by 3-CFG, 4-CFG, etc...
- What about SCFGs?
  - Rank 3:

$$\begin{aligned} A &\rightarrow B\ [C\ D], & [C\ D]\ B \\ A &\rightarrow B\ V, & V\ B \\ V &\rightarrow C\ D, & C\ D \end{aligned}$$

Yes, we can. 2-SCFG = 3-SCFG.

– Rank 4:

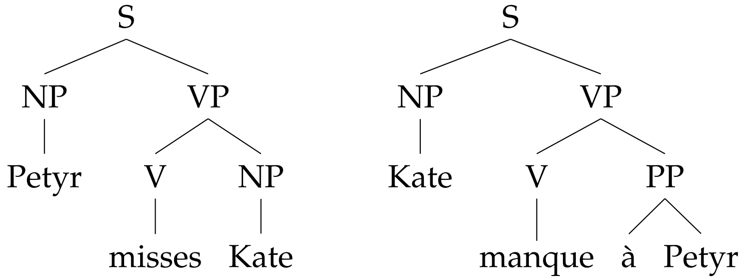
$$\begin{array}{ll}
A \to B\ C\ D\ E, & C\ E\ B\ D \\
A \to [B\ C]\ D\ E, & [C\ E\ B]\ D \\
A \to B\ [C\ D]\ E, & [C\ E\ B\ D] \\
A \to B\ C\ [D\ E], & C\ [E\ B\ D]
\end{array}$$

In each chunk that we might want to replace in the first language, we have one or more intervening symbols in the second language. Therefore, 3-SCFG $\subsetneq$ 4-SCFG.

– The subset of 2-SCFG = 3-SCFG is equivalently called **inversion transduction grammar**. The notation is slightly different, we write $A \to [B\ C]$ when the order is preserved and $A \to \langle B\ C \rangle$ when it is inverted.

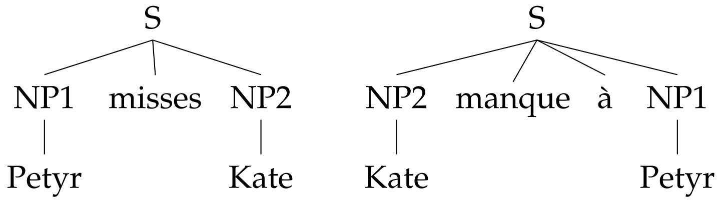
No raising or lowering

SCFGs can only reorder sibling nodes. Is that enough? Not always.



SCFGs cannot swap the subject and object, because they aren't siblings in the original grammar.

We could solve this by changing the grammar,



By including the verb *misses/manque à* directly into the rule, we ensure that it doesn't apply to other verbs.

With other syntactic translation models (synchronous tree substitution grammar or tree adjoining grammars), this case can be handled without flattening.

19.5 Algorithms for SCFGs

Translation

In principle, translation in SCFGs is nearly identical to parsing. Suppose we have the Spanish phrase *la razón principal*, and the synchronous grammar

$$
\begin{array}{rlrl}
NP \to & D\ NPB, & D\ NPB & 1.0 \\
NPB \to & N_1\ J_2, & J_2\ N_1 & 0.8 \\
NPB \to & N_1\ N_2, & N_1\ N_2 & 0.2 \\
D \to & \textit{la}, & \textit{the} & 0.5 \\
N \to & \textit{razon}, & \textit{reason} & 0.5 \\
N \to & \textit{principal}, & \textit{principal} & 0.5 \\
J \to & \textit{principal}, & \textit{main} & 1.0
\end{array}
$$

Now we can apply CKY, building the translation on the English side. We should get two possible translations, *the reason principal* ($p(e, f, \tau) = 0.05$) and *the main reason* ($p(e, f, \tau) = 0.4$).

What is the complexity of translation with binarizable SCFGs? It's just like CFG parsing: $\mathcal{O}(n^3)$.

Bitext parsing

To learn a translation model, we might need to synchronously parse the **bitext**: both the source and target side language.

We can do this with a dynamic program.

Assuming we are dealing with 2-SCFG or 3-SCFG, here's what we need to keep track of:

- The non-terminals that we have derived
- Their spans in the source language (start and end)
- Their spans in the target language (start and end)

Suppose we are given spans $\langle i, j\rangle$ in the source and $\langle i', j'\rangle$ in the target. Then we are looking for split points k and k' and a production that can derive the subspans $\langle i, k\rangle, \langle k, j\rangle$ and $\langle i', k'\rangle, \langle k', j'\rangle$.

What is the space complexity of bitext parsing? $\mathcal{O}(|S|n^4)$, where $|S|$ is the number of non-terminals.

What is the time complexity of bitext parsing? $\mathcal{O}(|R|n^6)$, where $|R|$ is the number of production rules.

Specificially, we have the recurrence

$$\begin{aligned}\psi(X,i,j,i',j') = \max_{k,k',A,B} & P(S \to A\ B, A\ B) \otimes \psi(A,i,k,i',k') \otimes \psi(B,k,j,k',j') \\ & \oplus P(S \to A\ B, B\ A) \otimes \psi(A,i,k,k',j') \otimes \psi(B,k,j,i',k')\end{aligned}$$

Note: in general, bitext parsing is exponential in the rank of the SCFG (unless $P = NP$).

Intersection with language model

For fluent translations, we typically want to multiply in the language model probability on the target side.

- This (usually) corresponds **intersection** of an SCFG with a finite state machine.
- Sidenote: what about context-free language models?
  - $A = \{a^m b^m c^n\}$
  - $B = \{a^m b^n c^n\}$
  - $A \cap B = \{a^n b^n c^n\}$, not a CFL!
  - CFLs are not closed under intersection.
  - Determining if $s \in A \cap B$ is in PSPACE
- There are exact dynamic programming algorithms for intersecting an SCFG and an FSA, but they are very slow. One solution is **cube pruning**.
- We can equivalently view this as an ILP

$$\begin{aligned} min. \quad & \sum_v \theta_v y_v + \sum_e \theta_e y_e + \sum_{\langle v,w\rangle \in \mathcal{B}} \theta(v,w) y(v,w) \\ s.t. \quad & C0 : y_v, y_e \text{ form a derivation} \\ & C1 : y_v = \sum_{w:\langle w,v\rangle \in \mathcal{B}} y(w,v) \\ & C2 : y_v = \sum_{w:\langle v,w\rangle \in \mathcal{B}} y(v,w) \end{aligned}$$

- Here y_e and y_v are indicator variables that define what words and hyperedges appear in the derivation.
- We can solve this optimization with Lagrangian relaxation.
  - Replace the outgoing constraints $C2$ with multipliers $u(v)$
  - At first, $u(v) = 0, \forall v$
  - Without the outgoing constraints, we can optimize efficiently
  - If the outgoing constraints happen to be met, we are done
  - Otherwise, update $u(v)$ and try again.
- Lagrangian relaxation finds the exact solution 97% of the time, is many times faster than ILP.

Part V

Learning

Chapter 20

Semi-supervised learning

So far we have focused on learning a classifier — typically represented by a set of weights $\boldsymbol{\theta}$ — from a set of labeled examples $\{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^N$. As we have seen, it is possible to formulate structured prediction tasks such as parsing in this same framework. But what if you don't have those labeled examples for the domain or task that you want to solve?

This scenario happens all the time — class projects, interdisciplinary collaborations, and commercial applications. As text is increasingly available online (social media, patient medical records, e-government), there are more and more datasets that could be fodder for NLP. Lack of labeled data in the target domains and tasks is the main limitation to language technology being applicable more broadly.

There are two "simple" solutions that one might undertake:

1. Use some other labeled data and hope it works.

   Unfortunately, it probably won't. For example, in applying parsers trained on the Penn Treebank to social media texts, researchers have observed massive decreases in accuracy (Foster et al., 2011; Gimpel et al., 2011).

2. Label data yourself.

   This is a lot of work. For example:

   - **The Switchboard corpus** contains phoneme annotations of telephone conversations, e.g.

     film → F IH_N UH_GL_N M

     be all → BCL B IY IY_TR AO_TR AO L_DL

     This took 400 hours of annotation time per hour of speech.

- **The Penn Chinese Treebank** is a set of CFG annotations for Chinese. It took 2 years to get 4000 sentences annotated.

Crowd-sourcing has recently become popular as a means to annotate large amounts of data quickly. This can work well, but effort and expertise and needed to get good annotations for linguistically complex tasks (Snow et al., 2008; Zaidan and Callison-Burch, 2011).

In this chapter, we will explore an alternative to either of these approaches: harnessing data that is unlabeled, or is labeled in a different domain or task. We will think of our annotated data as a *sample* from some underlying distribution.

$$\{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^{N} \sim \mathcal{D} \tag{20.1}$$

This allows us to formulate various learning scenarios:

Semisupervised learning Imagine that N is small, so that it is hard to learn a model that generalizes well. We would like to leverage **unlabeled** data,

$$\{\boldsymbol{x}_i\}_{i=1}^{N_u} \sim \mathcal{D}, \tag{20.2}$$

which is drawn from the same underlying distribution $\mathcal{D}$, but for which labels are unavailable. Since this data is not labeled, it is usually available in very large quantities, so $N_u \gg N$.

We have already seen two examples of semi-supervised learning. The first was the use of **expectation-maximization** in document classification in chapter 4; in the E-step, we impute beliefs about the labels of unlabeled documents, and then use these beliefs to update our model in the M-step (Nigam and Ghani, 2000). Another example of semi-supervised learning was given in chapter 15. There we saw how to use unlabeled data to build **Brown clusters**. These clusters then act as features, generalizing over individual words by capturing lexical similarity (Miller et al., 2004; Koo et al., 2008).

While these techniques are effective, they are limited. Expectation-maximization requires a generative model, which may be a less effective classifier than a discriminative alternative such as logistic regression or support vector machines. Brown clusters are useful features, but they are learned separately from the main label prediction task. In **??**, we will explore additional techniques for semisupervised learning, such as bootstrapping and multiview learning.

Active learning This setting is similar to semi-supervised learning, but with a twist: we can iteratively query a user for labels for a small number of unlabeled instances. This is relevant in commercial settings, where a company can pay a small staff of annotators to label examples until performance is good enough. The key question is deciding which examples to label next. Settles (2010) surveys a number of alternatives; we will not explore the issue here.

Supervised domain adaptation Now imagine that we have a large amount of labeled data in some **source** domain, but a much smaller amount of information in the **target** domain. For example, the source domain could be 20th century newspaper articles (as in the Penn Treebank), and the target domain could be something like social media posts or patient medical records. We don't have enough target domain data to learn a good model. But if we simply combine all the data from the two domains, the source domain instances will dominate, and we will suffer from the resulting **domain shift**. We will consider various techniques for learning effectively from both domains.

Multitask (transfer) learning Similar to supervised domain adaptation, but rather than assuming that the underlying distribution $P(X, Y)$ shifts across domains, we assume that only the label distribution $P(Y \mid X)$ shifts. For example, we are working in the newstext domain, and we have a large amount of labeled data for part-of-speech tagging, and a small amount of labeled data for named-entity recognition. The goal is then to learn a better model using both labeled datasets.

Unsupervised domain adaptation This setting combines features of semisupervised learning and supervised domain adaptation: we have labeled data in the source domain, but no labeled data in the target domain. The prototypical example of this situation is in sentiment polarity analysis of product reviews: you are given annotated reviews of, say, coffee machines, but you want to predict the sentiment for reviews of bicycles (Blitzer et al., 2007). Another relevant setting is the application of syntactic analyzers such as part-of-speech taggers to historical texts (Yang and Eisenstein, 2015).

20.1 Semisupervised learning

Let's first consider the question of why would unlabeled data might help in a supervised classification task. Suppose you want to do sentiment analysis in French.

I give you two labeled examples:

(20.1) ☺ *émouvant avec grâce et* ***style***

(20.2) ☹ *fastidieusement inauthentique et* ***banale***

You have a bunch of unlabeled examples too:

(20.3) *pleine de* ***style*** *et d'****intrigue***

(20.4) *la* ***banalité*** *n'est dépassée que par sa* ***prétention***

(20.5) ***prétentieux****, de la première minute au rideau final*

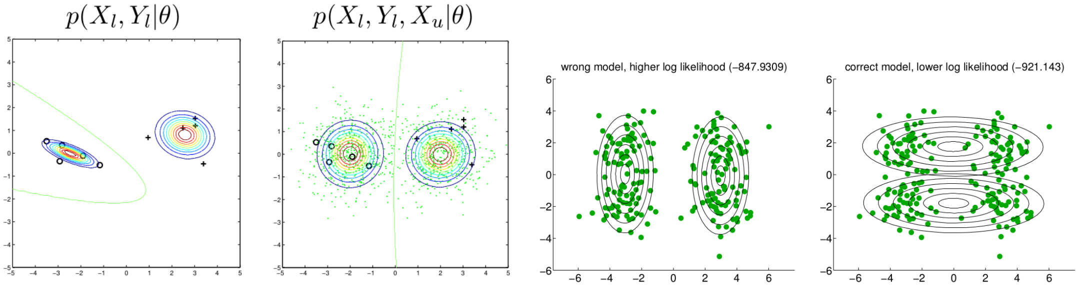
(20.6) *imprégné d'un air d'****intrigue***

If we just learn from the labeled data, we might conclude that *style* is positive and that *banale* is negative. This isn't much. However, we can propagate this information to the unlabeled data, and potentially learn more.

- If we are confident about *style* being positive, then we can guess that (20.3) is also positive.
- That suggests that *intrigue* is also positive.
- We can then propagate this information to (20.6), and learn more.
- Similarly, we can propagate from the labeled data to (20.4), which we guess to be negative. This suggests that *pretention* is also negative, which we propagate to (20.5).

What happened here? Instances (20.3) and (20.4) were "similar" to our labeled examples for positivity and negativity, respectively. We used them to expand those concepts, which allowed us to correctly label instances (20.5) and (20.6), which didn't share any important features with our original labeled data. In doing this, we made a key assumption: that similar instances will have similar labels. (Is this assumption reasonable? Keep this question in mind.) In this case, we defined similarity in terms of sharing some key words (non-stopwords).

To see how this can help conceptually, think about similarity just in terms of 1D space. If you have only the two labeled instances, your decision boundary should be right in between. (Do you remember what criterion justifies this choice?) But if you have a bunch of unlabeled instances, you might want to draw this boundary in a different place. Let's now see how we can operationalize this idea in an algorithm.



(a) Now it works . . .

(b) now it doesn't.

Figure 20.1: Expectation-maximization for semi-supervised learning on Gaussian data [todo: find credits for these images; Jerry Zhu?]

Semi-supervised learning with EM

We've already seen one way to do this: use expectation-maximization (EM) to marginalize over the labels of the unseen data. So we are maximizing

$$\mathrm{p}(X^\ell, Y^\ell, X^U) = \mathrm{p}(X^\ell, Y^\ell) \sum_{Y^U} \mathrm{p}(X^U, Y^U). \tag{20.3}$$

Expectation-maximization maximizes a bound on the joint probability defined above, by iterating between two steps:

E-step Fit a distribution $Q(y_i)$ for all unlabeled i;

M-step Maximize the expected likelihood under this distribution.

You can see why this can work in the example shown in Figure 20.1a: by incorporating unlabeled data, we get a much more reasonable decision boundary.

However, things can also go wrong, as shown in Figure 20.1b. In this example, the correct model (left) has a lower log-likelihood than the incorrect model (right). The basic problem here is that the model is wrong. The label is related to the observations, but not in the simplistic, Gaussian way that we had assumed. In chapter 4, we discussed a heuristic to try to deal with this problem: downweighting the contribution of the unseen data to the likelihood function. But this requires setting the weight parameter, which depends on a host of problem-specific characteristics, such as the underyling variance of the data. We will now consider some alternatives that often work better.

Bootstrapping and co-training

EM is sort of like self-training or **bootstrapping**: we use our current model to estimate $Q(y_i)$, and then update the model as if $Q(y_i)$ is correct.

- The probabilistic nature of this is nice, but it limits us to relatively weak classifiers.
- If we are willing to give up on probability, we can bootstrap **any** classifier.

Rather than imputing beliefs about all unlabeled instances $Q(y_i)$, we can add just a few, highly confident instances at each step. This is similar to how we proceeded in the French sentiment labeling example above. The simplest version of this algorithm is 1-nearest-neighbor: for each unlabeled data point, if its nearest neighbor has a label, then propagate that label. This approach does not make the parametric assumptions that doomed us in Figure 20.1b; instead, it relies on the similarity graph over instances. For some types of data, this is more reasonable, but it can also fail, as shown in the slides [todo: add these figures here].

There is some "folk wisdom" about when bootstrapping works:

- It works better for generative models (e.g., Naive Bayes) than for discriminative models (e.g., perceptron)
- It works better when the Naive Bayes assumption is stronger.
    - Suppose we want to classify NEs as PERSON or LOCATION
    - Features: string and context
        - *located on Peachtree Street*
        - *Dr. Walker said ...*

$$
\begin{aligned}
&P(W_{m+1} = \textit{street}, W_{m-1} = \textit{on} \mid Y_m = \textsc{loc}) \\
&\quad \approx P(W_{m+1} = \textit{street} \mid Y_m = \textsc{loc}) P(W_{m-1} = \textit{on} \mid Y_m = \textsc{loc})
\end{aligned}
$$

Cotraining makes the bootstrapping assumptions explicit (Blum and Mitchell, 1998).

- Assume two, **conditionally independent**, views of a problem.
- Assume each view is sufficient to do good classification.

Sketch of learning algorithm:

- On labeled data, minimize error.
- On unlabeled data, **constrain** the models from different views to agree with each other.

Co-training example See the slides for an animated version of this. Assume we want to do named entity classification: determine whether an NE is a Location or Person. We have two views: the name itself, and its context.

	$\boldsymbol{x}^{(1)}$	$\boldsymbol{x}^{(2)}$	y
1.	Peachtree Street	located on	LOC
2.	Dr. Walker	said	PER
3.	Zanzibar	located in	? → LOC
4.	Zanzibar	flew to	? → LOC
5.	Dr. Robert	recommended	? → PER
6.	Oprah	recommended	? → PER

Algorithm

- Use classifier 1 to label example 5.
- Use classifier 2 to label example 3.
- Retrain both classifiers, using newly labeled data.
- Use classifier 1 to label example 4.
- Use classifier 2 to label example 6.

Multiview Learning is another approach in this style. Cotraining treats the output of each view's classifier as a labeled instance for the other view. In multiview learning, we add a **co-regularizer** that penalizes disagreement between the views on the unlabeled instances. This allows us to define a single objective function. In the case of two-view linear regression, the function is

$$\begin{aligned}\min_{w,v} \sum_{i}^{L} (y_i - \boldsymbol{w}^\top \boldsymbol{x}_i^{(1)})^2 + (y_i - \boldsymbol{v}^\top \boldsymbol{x}_i^{(2)})^2 + \lambda_1 ||\boldsymbol{w}||^2 + \lambda_1 ||\boldsymbol{v}||^2 \\ + \lambda_2 \sum_{i=L+1}^{L+U} (\boldsymbol{w}^\top \boldsymbol{x}_i^{(1)} - \boldsymbol{v}^\top \boldsymbol{x}_i^{(2)})^2 \qquad (20.4)\end{aligned}$$

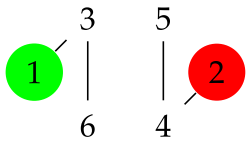


Figure 20.2: Semi-supervised sentiment analysis as a graph

The only difference from standard regression is the co-regularizer, which penalizes disagreement on the unlabeled data.

An early version of this idea is **co-boosting** (Collins and Singer, 1999), where each view is a boosting classifier, and features are added incrementally to each view.

Graph-based approaches

Let's go back to sentiment analysis in French. We can view this data as a **graph**, with edges between similar instances, as shown in Figure 20.2. Unlabeled instances propagate information through the graph.

Where does the graph come from?

- Sometimes there is a natural similarity metric (time, position in the document).
- Otherwise, we can compute similarity from features. If the features are Gaussian, we could say:

  $$\text{sim}(i,j) = \exp\left(-\frac{||\boldsymbol{x}_i - \boldsymbol{x}_j||^2}{2\sigma^2}\right)$$

  If the features are discrete, we might use KL-divergence.
- Then we add an edge between i and j when sim$(i,j) > \tau$

Given a graph with edge weights s_{ij}, we can formulate semi-supervised learning as an optimization problem:

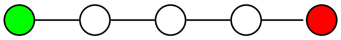
$$\begin{aligned} &\min_{\boldsymbol{z}} \sum_{i,j} s_{ij}(z_i - z_j)^2 \\ &s.t. \forall_{i \in \{1 \ldots N_\ell\}} z_i = y_i \\ &\forall_i z_i \in \{0, 1\} \end{aligned} \tag{20.5}$$

This looks like a combinatorial problem. Specifically, it looks like (binary) integer linear programming, which is NP-complete. But assuming $s_{ij} \geq 0$, this specific problem can be reformulated as maximum-flow, with polynomial time solutions. Rao and Ravichandran (2009) apply this idea to expanding polarity lexicons. In their graph:

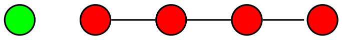
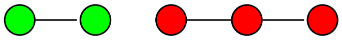
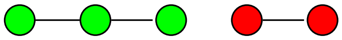
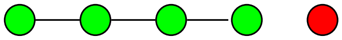
- Nodes are words
- Edges are wordnet relations
- They label a few nodes for sentiment polarity, and propagate those labels to other parts of the graph.
- However, they use a slightly modified version of the mincut idea: randomized min-cut (Blum et al., 2004).

Randomized min-cut

Suppose we have this initial graph:



What is the solution? Actually, the following solutions are all equivalent:

- 
- 
- 
- 

Another problem with mincuts is that it doesn't distinguish high-confidence and low-confidence predictions. Both of these problems can be dealt with by randomization:

- Add random noise to adjacency matrix.
- Rerun mincuts multiple times.
- Deduce the final classification by voting.

Label propagation

A related approach is **label propagation** (Zhu and Ghahramani, 2002), which Rao and Ravichandran also consider. The basic idea is that we relax y_i from an integer $\{0, 1\}$ to a real number $\mathbb{R}$. Then we solve the optimization problem,

$$\begin{aligned}&\min_Y \sum_{i,j} s_{ij}(y_i - y_j)^2\\ &s.t. Y_L \text{ is clamped to initial values}\end{aligned}$$

The advantages are:

- a unique global optimum
- a natural notion of confidence: distance of y_i from 0.5

Let's look at the objective:

$$\begin{aligned}J =&\frac{1}{2}\sum_{i,j} s_{ij}(y_i - y_j)^2\\ =&\frac{1}{2}\sum_{i,j} s_{ij}(y_i^2 + y_j^2 - 2y_iy_j)\\ =&\sum_i y_i^2 \sum_j s_{i,j} - \sum_{i,j} s_{ij}y_iy_j\\ =&\boldsymbol{y}^\top \mathbf{D}\boldsymbol{y} - \boldsymbol{y}^\top \mathbf{S}\boldsymbol{y}\\ =&\boldsymbol{y}^\top \mathbf{L}\boldsymbol{y}\end{aligned}$$

We have introduced three matrices

- Let $\mathbf{S}$ be the $n \times n$ similarity matrix.
- Let $\mathbf{D}$ be the **degree matrix**, $d_{ii} = \sum_j s_{ij}$. $\mathbf{D}$ is diagonal.
- Let $\mathbf{L}$ be the unnormalized **graph Laplacian** $\mathbf{L} = \mathbf{D} - \mathbf{S}$
- So we want to minimize $\boldsymbol{y}^\top \mathbf{L}\boldsymbol{y}$ with respect to $\boldsymbol{y}_u$, the labels of the unannotated instances.

In principle, this is easily solveable:

- Partition the Laplacian $\mathbf{L} = \begin{bmatrix}\mathbf{L}_{\ell\ell} & \mathbf{L}_{\ell u}\\ \mathbf{L}_{u\ell} & \mathbf{L}_{uu}\end{bmatrix}$

- Then the closed form solution is $\boldsymbol{y}_u = -\mathbf{L}_{uu}^{-1}\mathbf{L}_{u\ell}\boldsymbol{y}_\ell$
- This is great ... if we can invert $\mathbf{L}_{uu}$.

In practice, $\mathbf{L}_{u,u}$ is huge, so we can't invert it unless it has special structure. Zhu and Ghahramani (2002) propose an iterative solution called **label propagation**.

- Let $\mathbf{T}_{ij} = \frac{s_{ij}}{\sum_k s_{kj}}$, row-normalizing $\mathbf{S}$.
- Let $\mathbf{Y}$ be an $n \times C$ matrix of labels, where C is the number of classes.
- Until tired,
  - Set $\mathbf{Y} = \mathbf{TY}$
  - Row-normalize $\mathbf{Y}$
  - Clamp the seed examples in $\mathbf{Y}$ to their original values
- There's a flavor of EM here, with $\mathbf{Y}$ representing our belief $q_i(y_i)$. But there's no M-step in which we update model parameters. That's because we're in a graph-based framework, and we're assuming the graph is correct.

Both mincut and label propagation are **transductive** learning algorithms: they learn jointly over the training and test data. This is fine in some settings, but not if you want to train a system and then apply it to new test data later — you'd have to retrain it all over again.

Manifold regularization (Belkin et al., 2006) addresses this issue, by learning functions that are smooth on the "graph manifold." In practice, this means that they give similar labels to nearby datapoints in the unlabeled data. Suppose we are interested in learning a classification function f. Then we can optimize:

$$\text{argmin}\, f \frac{1}{\ell} \sum_i \ell(f(\boldsymbol{x}_i), y_i) + \lambda_1 ||f||^2 + \lambda_2 \sum_{i,j} s_{ij}(f(\boldsymbol{x}_i) - f(\boldsymbol{x}_j))^2$$

- The first term corresponds to the loss on the labeled training data; we can use any convex loss functions, such as logistic or hinge loss.
- The second term corresponds to the smoothness, akin to regularizing the weights in a linear classifier.
- The third term penalizes making different predictions for similar instances in the unlabeled data

The representer theorem guarantees that we can solve for f as long as ℓ is convex. We can then apply f to any new unlabeled test data.

20.2 Domain adaptation

In domain adaptation, we have a lot of labeled data, but it's in the wrong domain. Some features will be shared across domains. For example, if we are classifying movies or toasters, *good* is a good word, and *sucks* is a bad word. But as we've seen, real review text is usually more subtle. What about a word like *unpredictable*? This is a good word for a movie, but not such a good word for a kitchen appliance.

Supervised domain adaptation

In supervised domain adaptation (transfer learning), we have:

- Lots of labeled data in a "source" domain, $\{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^{\ell_S} \sim \mathcal{D}_S$ (e.g., reviews of restaurants)
- A little labeled data in a "target" domain, $\{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^{\ell_T} \sim \mathcal{D}_T$ (e.g., reviews of chess stores)

Here are some (surprisingly-competitive) baselines (see slides)

- Source-only: train on the source data, apply it to the target data.
- Target-only: forget the source data, just train on the limited target data.
- Big blob: merge the source and target data into a single training set. Optionally downweight the source data.
- Prediction: train a classifier on the source data, use its prediction as a feature in the target data.
- Interpolation: train two classifiers, combine their outputs

Here are two less-obvious approaches:

Priors :
Train a (logistic-regression) classifier on the source data. Treat its weights as the priors on the target data, and regularize towards these weights rather than towards zero (Chelba and Acero 2004).

Feature augmentation Create **copies** of each feature, for each domain and for the cross-domain setting.

- The copies fire in the appropriate domains, and the learning algorithm decides whether a feature is general or domain-specific.

- For example, suppose we have domains for Appliances and Movies, and features *outstanding* and *sturdy*. We replicate the features, obtaining

$$\langle \textit{outstanding}, \text{APP.} \rangle, \langle \textit{outstanding}, \text{MOV.} \rangle, \langle \textit{outstanding}, \text{ALL} \rangle$$
$$\langle \textit{sturdy}, \text{APP.} \rangle, \langle \textit{sturdy}, \text{MOV.} \rangle, \langle \textit{sturdy}, \text{ALL} \rangle$$

- Ideally, we will learn a positive weight for $\langle \textit{outstanding}, \text{ALL} \rangle$, because the feature works in both domains, and a small weight for the domain-specific copies of the *outstanding* feature.
- We will also learn a positive weight for $\langle \textit{sturdy}, \text{APP} \rangle$, because the feature works only in the Appliance domain.

See slides for a diagram of how this works.

Unsupervised domain adaptation

Without labeled data in the target domain, can we learn anything? If the source and target domain are somewhat related, then we can. A very popular approach is structural correspondence learning (SCL) (Blitzer et al., 2007).

- Suppose there are a few words that are good predictors in both domains; we'll call these **pivot features**
- Pivot features can be selected by finding words that are
  - Popular in both domains
  - High mutual-information with the label in the source domain
- The label is unknown in the target domain, so we can't learn to predict it. Instead we'll predict the pivots. We train a linear classifier for each pivot, obtaining weights $\boldsymbol{\theta}_n$ for pivot n.
- For example, we can learn that the domain-specific feature *fast-multicore* is a good predictor of the pivot *excellent*.
- We can horizontally concatenate the pivot predictor weights, forming

$$\Theta = [\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \ldots, \boldsymbol{\theta}_N] \tag{20.6}$$

- The matrix Θ is large, and contains redundant information (since many pivots are closely related to each other). We factor $\Theta \approx USV^T$ using singular value decomposition (SVD).

- We use U to **project** features from both domains into a shared space, $U^\top \boldsymbol{x}$.
- We then learn to predict the label in the source domain, using the augmented instance $\langle \boldsymbol{x}, U^\top \boldsymbol{x} \rangle$. In U contains meaningful correspondences between the domains, then the weights learned on these features will work for the target domain instances too.
- This idea yields substantial improvements in adapting sentiment classifiers across product domains, e.g., books, movies, and appliances (Blitzer et al., 2007).

See the slides for a graphical explanation of these ideas, with slightly different notation.

20.3 Other learning settings

There are many other settings in which we learn from something other than in-domain labeled data:

- **Active learning**. The model can query the annotator for labels (see above)
- **Feature labeling**. Annotators label *features* rather than instances. For example, you provide a list of five prototype words for each POS tag (Haghighi and Klein, 2006).
- **Feature expectations**. Learn from *constraints* on feature-label relationships; for example, the word "the" is a determiner at least 90% of the time. In EMNLP 2013, this idea was applied to multilingual learning (which I'll discuss in the final lecture). The basic idea of this paper is to align words between sentences and insist that aligned words have the same tag most of the time.
- **Multi-instance learning**. The learner gets a "bag" of instances, and a label. If the label is positive, then at least one instance in the bag is positive, but you don't know which one.

  This idea is often related to **distant supervision**. The learner gets a label indicating that there is a relationship, such as $\textsc{Born-in}(\textsc{Obama}, \textsc{Hawaii})$, and a set of instances containing sentences that mention the two arguments, *Obama* and *Hawaii*. Many of these sentences do not actually instantiate the desired relation (e.g., *Obama visited Hawaii in 2008...*), but we assume that at least one does, and we must learn from this.

Chapter 21

Beyond linear models

21.1 Representation learning

21.2 Convolutional neural networks

21.3 Recursive neural networks

21.4 Encoder-decoder models

21.5 Structure prediction

Recently, several researchers have applied neural networks and other distributed representations to dependency parsing. These methods diverge from the approach of scoring edges by the inner product of a weight vector with a large, sparse feature vector. Instead, each word is represented by a small, dense **embedding** vector, which may be estimated from unlabeled data in a preprocessing step. These embeddings are typically used in combination with transition-based dependency parsers, either as features (Bansal et al., 2014), or as part of an integrated neural network parsing model (Henderson et al., 2008; Chen and Manning, 2014; Dyer et al., 2015). These models are described in more detail in chapter 21. Embeddings can also be learned for features (rather than for words) in a graph-based parsing algorithm (Lei et al., 2014).

Bibliography

Abney, S. P. and Johnson, M. (1991). Memory requirements and local ambiguities of parsing strategies. *Journal of Psycholinguistic Research*, 20(3):233–250.

Akaike, H. (1974). A new look at the statistical model identification. *Automatic Control, IEEE Transactions on*, 19(6):716–723.

Akmajian, A., Demers, R. A., Farmer, A. K., and Harnish, R. M. (2010). *Linguistics: An introduction to language and communication*. MIT press, Cambridge, MA, sixth edition.

Allauzen, C., Riley, M., and Schalkwyk, J. (2009). A generalized composition algorithm for weighted finite-state transducers. In *INTERSPEECH*.

Allauzen, C., Riley, M., Schalkwyk, J., Skut, W., and Mohri, M. (2007). Openfst: A general and efficient weighted finite-state transducer library. In *International Conference on Implementation and Application of Automata*, pages 11–23. Springer.

Anandkumar, A., Ge, R., Hsu, D., Kakade, S. M., and Telgarsky, M. (2014). Tensor decompositions for learning latent variable models. *The Journal of Machine Learning Research*, 15(1):2773–2832.

Andreas, J. and Klein, D. (2015). When and why are log-linear models self-normalizing? In *Proceedings of the North American Chapter of the Association for Computational Linguistics (NAACL)*, pages 244–249, Denver, CO.

Arora, S., Ge, R., Halpern, Y., Mimno, D., Moitra, A., Sontag, D., Wu, Y., and Zhu, M. (2013). A practical algorithm for topic modeling with provable guarantees. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 280–288.

Bansal, M., Gimpel, K., and Livescu, K. (2014). Tailoring continuous word representations for dependency parsing. In *Proceedings of the Association for Computational Linguistics (ACL)*, Baltimore, MD.

Belkin, M., Niyogi, P., and Sindhwani, V. (2006). Manifold regularization: A geometric framework for learning from labeled and unlabeled examples. *The Journal of Machine Learning Research*, 7:2399–2434.

Bender, E. M. (2013). *Linguistic Fundamentals for Natural Language Processing: 100 Essentials from Morphology and Syntax*, volume 6 of *Synthesis Lectures on Human Language Technologies*. Morgan & Claypool Publishers.

Bengio, Y., Ducharme, R., Vincent, P., and Janvin, C. (2003). A neural probabilistic language model. *The Journal of Machine Learning Research*, 3:1137–1155.

Berg-Kirkpatrick, T., Bouchard-Côté, A., DeNero, J., and Klein, D. (2010). Painless unsupervised learning with features. In *Proceedings of the North American Chapter of the Association for Computational Linguistics (NAACL)*, pages 582–590, Los Angeles, CA.

Berger, A. L., Pietra, V. J. D., and Pietra, S. A. D. (1996). A maximum entropy approach to natural language processing. *Computational linguistics*, 22(1):39–71.

Bergsma, S., Lin, D., and Goebel, R. (2008). Distributional identification of non-referential pronouns. In *Proceedings of the Association for Computational Linguistics (ACL)*, pages 10–18, Columbus, OH.

Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D., and Bengio, Y. (2010). Theano: A cpu and gpu math compiler in python. In *Proceedings of the 9th Python in Science Conf*, pages 1–7.

Bikel, D. M. (2004). Intricacies of Collins' parsing model. *Computational Linguistics*, 30(4):479–511.

Bishop, C. M. (2006). *Pattern recognition and machine learning*. springer.

Blackburn, P. and Bos, J. (2005). *Representation and inference for natural language: A first course in computational semantics*. CSLI.

Blei, D. M. (2012). Probabilistic topic models. *Communications of the ACM*, 55(4):77–84.

Blei, D. M. (2014). Build, compute, critique, repeat: Data analysis with latent variable models. *Annual Review of Statistics and Its Application*, 1:203–232.

Blitzer, J., Dredze, M., and Pereira, F. (2007). Biographies, bollywood, boom-boxes and blenders: Domain adaptation for sentiment classification. In *Proceedings of the Association for Computational Linguistics (ACL)*, pages 440–447, Prague.

Blum, A., Lafferty, J., Rwebangira, M. R., and Reddy, R. (2004). Semi-supervised learning using randomized mincuts. In *icml*, page 13.

Blum, A. and Mitchell, T. (1998). Combining labeled and unlabeled data with co-training. In *Proceedings of the eleventh annual conference on Computational learning theory*, pages 92–100. ACM.

Böhmová, A., Hajič, J., Hajičová, E., and Hladká, B. (2003). The prague dependency treebank. In *Treebanks*, pages 103–127. Springer.

Bottou, L. (1998). Online learning and stochastic approximations. *On-line learning in neural networks*, 17:9.

Bottou, L., Curtis, F. E., and Nocedal, J. (2016). Optimization methods for large-scale machine learning. *arXiv preprint arXiv:1606.04838*.

Boyd, S. and Vandenberghe, L. (2004). *Convex Optimization*. Cambridge University Press, New York, NY, USA.

Briscoe, T. (2011). Introduction to formal semantics for natural language.

Brody, S. and Diakopoulos, N. (2011). Cooooooooooooooollllllllllllll!!!!!!!!!!!!!!: using word lengthening to detect sentiment in microblogs. In *Proceedings of Empirical Methods for Natural Language Processing (EMNLP)*, pages 562–570.

Brown, P. F., Desouza, P. V., Mercer, R. L., Pietra, V. J. D., and Lai, J. C. (1992). Class-based n-gram models of natural language. *Computational linguistics*, 18(4):467–479.

Cappé, O. and Moulines, E. (2009). On-line expectation–maximization algorithm for latent data models. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 71(3):593–613.

Carreras, X., Collins, M., and Koo, T. (2008). Tag, dynamic programming, and the perceptron for efficient, feature-rich parsing. In *Proceedings of the Twelfth*

Conference on Computational Natural Language Learning, pages 9–16. Association for Computational Linguistics.

Carroll, L. (1917). *Through the looking glass: And what Alice found there*. Rand, McNally.

Charniak, E. (1997). Statistical techniques for natural language parsing. *AI magazine*, 18(4):33.

Charniak, E. (2000). A maximum-entropy-inspired parser. In *naacl*, pages 132–139.

Charniak, E. and Johnson, M. (2005). Coarse-to-fine n-best parsing and maxent discriminative reranking. In *Proceedings of the Association for Computational Linguistics (ACL)*, pages 173–180, Ann Arbor, Michigan.

Chen, D. and Manning, C. D. (2014). A fast and accurate dependency parser using neural networks. In *Proceedings of Empirical Methods for Natural Language Processing (EMNLP)*, pages 740–750.

Chen, S. F. and Goodman, J. (1999). An empirical study of smoothing techniques for language modeling. *Computer Speech & Language*, 13(4):359–393.

Cho, K. (2015). Natural language understanding with distributed representation. *CoRR*, abs/1511.07916.

Chu, Y.-J. and Liu, T.-H. (1965). On shortest arborescence of a directed graph. *Scientia Sinica*, 14(10):1396.

Chung, J., Gulcehre, C., Cho, K., and Bengio, Y. (2015). Gated feedback recurrent neural networks. In *Proceedings of the International Conference on Machine Learning (ICML)*.

Clarke, J., Goldwasser, D., Chang, M.-W., and Roth, D. (2010). Driving semantic parsing from the world's response. In *CONLL*, pages 18–27. Association for Computational Linguistics.

Collins, M. (1997). Three generative, lexicalised models for statistical parsing. In *Proceedings of the Association for Computational Linguistics (ACL)*, pages 16–23.

Collins, M. (2002). Discriminative training methods for hidden markov models: theory and experiments with perceptron algorithms. In *Proceedings of Empirical Methods for Natural Language Processing (EMNLP)*, pages 1–8.

Collins, M. (2003). Head-driven statistical models for natural language parsing. *Computational linguistics*, 29(4):589–637.

Collins, M. (2013). Notes on natural language processing. `http://www.cs.columbia.edu/~mcollins/notes-spring2013.html`.

Collins, M. and Brooks, J. (1995). Prepositional phrase attachment through a backed-off model. In *Workshop on Very Large Corpora*.

Collins, M. and Koo, T. (2005). Discriminative reranking for natural language parsing. *Computational Linguistics*, 31(1):25–70.

Collins, M. and Singer, Y. (1999). Unsupervised models for named entity classification. In *emnlp*, pages 189–196.

Collobert, R., Kavukcuoglu, K., and Farabet, C. (2011). Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, number EPFL-CONF-192376.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to algorithms*. MIT press, third edition.

Coviello, L., Sohn, Y., Kramer, A. D., Marlow, C., Franceschetti, M., Christakis, N. A., and Fowler, J. H. (2014). Detecting emotional contagion in massive social networks. *PloS one*, 9(3):e90315.

Crammer, K., Dekel, O., Keshet, J., Shalev-Shwartz, S., and Singer, Y. (2006). Online passive-aggressive algorithms. *The Journal of Machine Learning Research*, 7:551–585.

Crammer, K. and Singer, Y. (2001). Pranking with ranking. In *Neural Information Processing Systems (NIPS)*, pages 641–647, Vancouver.

Crammer, K. and Singer, Y. (2003). Ultraconservative online algorithms for multiclass problems. *The Journal of Machine Learning Research*, 3:951–991.

Cui, H., Sun, R., Li, K., Kan, M.-Y., and Chua, T.-S. (2005). Question answering passage retrieval using dependency relations. In *Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 400–407. ACM.

Culotta, A. and Sorensen, J. (2004). Dependency tree kernels for relation extraction. In *Proceedings of the Association for Computational Linguistics (ACL)*.

Das, D., Chen, D., Martins, A. F., Schneider, N., and Smith, N. A. (2014). Frame-semantic parsing. *Computational Linguistics*, 40(1):9–56.

Das, D., Martins, A. F., and Smith, N. A. (2012). An exact dual decomposition algorithm for shallow semantic parsing with constraints. In *Proceedings of SEMEVAL*, pages 209–217.

Das, D., Schneider, N., Chen, D., and Smith, N. A. (2010). Probabilistic frame-semantic parsing. In *Human language technologies: The 2010 annual conference of the North American chapter of the association for computational linguistics*, pages 948–956. Association for Computational Linguistics.

De Marneffe, M.-C., Dozat, T., Silveira, N., Haverinen, K., Ginter, F., Nivre, J., and Manning, C. D. (2014). Universal stanford dependencies: A cross-linguistic typology. In *LREC*, pages 4585–4592.

De Marneffe, M.-C., MacCartney, B., and Manning, C. D. (2006). Generating typed dependency parses from phrase structure parses. In *LREC*, pages 449–454.

De Marneffe, M.-C. and Manning, C. D. (2008). The stanford typed dependencies representation. In *Coling 2008: Proceedings of the workshop on Cross-Framework and Cross-Domain Parser Evaluation*, pages 1–8. Association for Computational Linguistics.

Deerwester, S. C., Dumais, S. T., Landauer, T. K., Furnas, G. W., and Harshman, R. A. (1990). Indexing by latent semantic analysis. *JASIS*, 41(6):391–407.

Derrida, J. (1985). Des tours de babel. In Graham, J., editor, *Difference in translation*. Cornell University Press, Ithaca, NY.

Dowty, D. (1991). Thematic proto-roles and argument selection. *Language*, pages 547–619.

Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12:2121–2159.

Durrett, G. and Klein, D. (2013). Easy victories and uphill battles in coreference resolution. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

Durrett, G. and Klein, D. (2015). Neural crf parsing. In *Proceedings of the Association for Computational Linguistics (ACL)*, Beijing.

Dyer, C. (2014). Notes on adagrad. `www.ark.cs.cmu.edu/cdyer/adagrad.pdf`.

Dyer, C., Ballesteros, M., Ling, W., Matthews, A., and Smith, N. A. (2015). Transition-based dependency parsing with stack long short-term memory. In *Proceedings of the Association for Computational Linguistics (ACL)*, pages 334–343, Beijing.

Edmonds, J. (1967). Optimum branchings. *Journal of Research of the National Bureau of Standards B*, 71(4):233–240.

Eisner, J. and Satta, G. (1999). Efficient parsing for bilexical context-free grammars and head automaton grammars. In *Proceedings of the 37th annual meeting of the Association for Computational Linguistics on Computational Linguistics*, pages 457–464. Association for Computational Linguistics.

Eisner, J. M. (1996). Three new probabilistic models for dependency parsing: An exploration. In *COLING*, pages 340–345.

Figueiredo, M., Graça, J., Martins, A., Almeida, M., and Coelho, L. P. (2013). LXMLS lab guide. `http://lxmls.it.pt/2013/guide.pdf`.

Fillmore, C. J. (1968). The case for case. In Bach, E. and Harms, R., editors, *Universals in linguistic theory*. Holt, Rinehart, and Winston.

Finkel, J. R., Grenager, T., and Manning, C. (2005). Incorporating non-local information into information extraction systems by gibbs sampling. In *Proceedings of the Association for Computational Linguistics (ACL)*, pages 363–370, Ann Arbor, Michigan.

Finkel, J. R., Grenager, T., and Manning, C. D. (2007). The infinite tree. In *Proceedings of the Association for Computational Linguistics (ACL)*, pages 272–279, Prague.

Finkel, J. R., Kleeman, A., and Manning, C. D. (2008). Efficient, feature-based, conditional random field parsing. In *Proceedings of the Association for Computational Linguistics (ACL)*, pages 959–967, Columbus, OH.

Firth, J. R. (1957). *Papers in Linguistics 1934-1951*. Oxford University Press.

Foster, J., Cetinoglu, O., Wagner, J., Le Roux, J., Nivre, J., Hogan, D., and van Genabith, J. (2011). From news to comment: Resources and benchmarks for parsing the language of web 2.0. In *Proceedings of the International Joint Conference* *on Natural Language Processing (IJCNLP)*, pages 893–901, Chiang Mai, Thailand. Asian Federation of Natural Language Processing.

Freund, Y., Schapire, R., and Abe, N. (1999). A short introduction to boosting. *Journal-Japanese Society For Artificial Intelligence*, 14(771-780):1612.

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

Fromkin, V., Rodman, R., and Hyams, N. (2013). *An introduction to language*. Cengage Learning.

Fundel, K., Küffner, R., and Zimmer, R. (2007). Relexrelation extraction using dependency parse trees. *Bioinformatics*, 23(3):365–371.

Gao, J., Andrew, G., Johnson, M., and Toutanova, K. (2007). A comparative study of parameter estimation methods for statistical natural language processing. In *Proceedings of the Association for Computational Linguistics (ACL)*, pages 824–831, Prague.

Ge, D., Jiang, X., and Ye, Y. (2011). A note on the complexity of l p minimization. *Mathematical programming*, 129(2):285–299.

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

Gimpel, K., Schneider, N., O'Connor, B., Das, D., Mills, D., Eisenstein, J., Heilman, M., Yogatama, D., Flanigan, J., and Smith, N. A. (2011). Part-of-speech tagging for Twitter: annotation, features, and experiments. In *Proceedings of the Association for Computational Linguistics (ACL)*, pages 42–47, Portland, OR.

Goldberg, Y. (2015). A primer on neural network models for natural language processing. *arXiv preprint arXiv:1510.00726*.

Goldwater, S. and Griffiths, T. (2007). A fully bayesian approach to unsupervised part-of-speech tagging. In *Annual meeting-association for computational linguistics*, volume 45.

Grosz, B. J., Weinstein, S., and Joshi, A. K. (1995). Centering: A framework for modeling the local coherence of discourse. *Computational linguistics*, 21(2):203–225.

Grosz, P. G., Patel-Grosz, P., Fedorenko, E., and Gibson, E. (2014). Constraints on donkey pronouns. *Journal of Semantics*, page ffu009.

Gundel, J. K., Hedberg, N., and Zacharski, R. (1993). Cognitive status and the form of referring expressions in discourse. *Language*, pages 274–307.

Gutmann, M. U. and Hyvärinen, A. (2012). Noise-contrastive estimation of unnormalized statistical models, with applications to natural image statistics. *The Journal of Machine Learning Research*, 13(1):307–361.

Haghighi, A. and Klein, D. (2006). Prototype-driven learning for sequence models. In *Proceedings of the North American Chapter of the Association for Computational Linguistics (NAACL)*, pages 320–327, New York, NY.

Haghighi, A. and Klein, D. (2009). Simple coreference resolution with rich syntactic and semantic features. In *Proceedings of Empirical Methods for Natural Language Processing (EMNLP)*, pages 1152–1161, Singapore.

Hannak, A., Anderson, E., Barrett, L. F., Lehmann, S., Mislove, A., and Riedewald, M. (2012). Tweetin'in the rain: Exploring societal-scale effects of weather on mood. In *Proceedings of the International Conference on Web and Social Media (ICWSM)*.

Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The elements of statistical learning*. Springer, New York, second edition.

Hatzivassiloglou, V. and McKeown, K. R. (1997). Predicting the semantic orientation of adjectives. In *Proceedings of the Association for Computational Linguistics (ACL)*, pages 174–181, Madrid, Spain.

Henderson, J., Merlo, P., Musillo, G., and Titov, I. (2008). A latent variable model of synchronous parsing for syntactic and semantic dependencies. In *CONLL*, pages 178–182.

Hindle, D. and Rooth, M. (1990). Structural ambiguity and lexical relations. In *Proceedings of the Workshop on Speech and Natural Language*.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

Hsu, D., Kakade, S. M., and Zhang, T. (2012). A spectral algorithm for learning hidden markov models. *Journal of Computer and System Sciences*, 78(5):1460–1480.

Hu, M. and Liu, B. (2004). Mining and summarizing customer reviews. In *Proceedings of Knowledge Discovery and Data Mining (KDD)*, pages 168–177.

Huang, L., Fayong, S., and Guo, Y. (2012). Structured perceptron with inexact search. In *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 142–151, Montréal, Canada. Association for Computational Linguistics.

Ide, N. and Wilks, Y. (2006). Making sense about sense. In *Word sense disambiguation*, pages 47–73. Springer.

Jaeger, T. and Levy, R. P. (2006). Speakers optimize information density through syntactic reduction. In *Neural Information Processing Systems (NIPS)*, pages 849–856.

Jaeger, T. F. (2010). Redundancy and reduction: Speakers manage syntactic information density. *Cognitive psychology*, 61(1):23–62.

Jain, A. K. (2010). Data clustering: 50 years beyond k-means. *Pattern recognition letters*, 31(8):651–666.

Jockers, M. L. (2015). Szuzhet? `http:bla.bla.com`.

Johnson, M. (1998). Pcfg models of linguistic tree representations. *Computational Linguistics*, 24(4):613–632.

Joshi, A. K. and Schabes, Y. (1997). Tree-adjoining grammars. In *Handbook of formal languages*, pages 69–123. Springer.

Jurafsky, D. (1996). A probabilistic model of lexical and syntactic access and disambiguation. *Cognitive Science*, 20(2):137–194.

Jurafsky, D. and Martin, J. H. (2009). *Speech and Language Processing*. Prentice Hall, 2 edition.

Karlsson, F. (2007). Constraints on multiple center-embedding of clauses. *Journal of Linguistics*, 43(02):365–392.

Khudanpur, S. and Wu, J. (2000). Maximum entropy techniques for exploiting syntactic, semantic and collocational dependencies in language modeling. *Computer Speech & Language*, 14(4):355–372.

Kilgarriff, A. (1997). I don't believe in word senses. *CoRR*, cmp-lg/9712006.

Kilgarriff, A. and Grefenstette, G. (2003). Introduction to the special issue on the web as corpus. *Computational linguistics*, 29(3):333–347.

Klein, D. and Manning, C. D. (2003). Accurate unlexicalized parsing. In *Proceedings of the Association for Computational Linguistics (ACL)*, pages 423–430.

Knight, K. and May, J. (2009). Applications of weighted automata in natural language processing. In *Handbook of Weighted Automata*, pages 571–596. Springer.

Koo, T., Carreras, X., and Collins, M. (2008). Simple semi-supervised dependency parsing. In *Proceedings of ACL-08: HLT*, pages 595–603, Columbus, Ohio. Association for Computational Linguistics.

Koo, T. and Collins, M. (2010). Efficient third-order dependency parsers. In *Proceedings of the Association for Computational Linguistics (ACL)*, pages 1–11, Uppsala, Sweden.

Koo, T., Globerson, A., Carreras, X., and Collins, M. (2007). Structured prediction models via the matrix-tree theorem. In *Proceedings of Empirical Methods for Natural Language Processing (EMNLP)*, pages 141–150.

Koren, Y., Bell, R., and Volinsky, C. (2009). Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Neural Information Processing Systems (NIPS)*, pages 1097–1105.

Kübler, S., McDonald, R., and Nivre, J. (2009). Dependency parsing. *Synthesis Lectures on Human Language Technologies*, 1(1):1–127.

Kuhlmann, M. and Nivre, J. (2010). Transition-based techniques for non-projective dependency parsing. *Northern European Journal of Language Technology (NEJLT)*, 2(1):1–19.

Kummerfeld, J. K., Berg-Kirkpatrick, T., and Klein, D. (2015). An empirical analysis of optimization for max-margin nlp. In *Proceedings of Empirical Methods for Natural Language Processing (EMNLP)*.

Lafferty, J., McCallum, A., and Pereira, F. (2001). Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *icml*.

Lari, K. and Young, S. J. (1990). The estimation of stochastic context-free grammars using the inside-outside algorithm. *Computer speech & language*, 4(1):35–56.

Lei, T., Xin, Y., Zhang, Y., Barzilay, R., and Jaakkola, T. (2014). Low-rank tensors for scoring dependency structures. In *Proceedings of the Association for Computational Linguistics (ACL)*, pages 1381–1391, Baltimore, MD.

Lesk, M. (1986). Automatic sense disambiguation using machine readable dictionaries: how to tell a pine cone from an ice cream cone. In *Proceedings of the 5th annual international conference on Systems documentation*, pages 24–26. ACM.

Levy, O. and Goldberg, Y. (2014). Neural word embedding as implicit matrix factorization. In *Neural Information Processing Systems (NIPS)*, Montréal.

Levy, R. and Manning, C. (2009). An informal introduction to computational semantics.

Liang, P., Jordan, M. I., and Klein, D. (2013). Learning dependency-based compositional semantics. *Computational Linguistics*, 39(2):389–446.

Liang, P. and Klein, D. (2009). Online em for unsupervised models. In *Proceedings of the North American Chapter of the Association for Computational Linguistics (NAACL)*, pages 611–619, Boulder, CO.

Liang, P., Petrov, S., Jordan, M. I., and Klein, D. (2007). The infinite pcfg using hierarchical dirichlet processes. In *Proceedings of Empirical Methods for Natural Language Processing (EMNLP)*, pages 688–697.

Lin, D. (1998). Automatic retrieval and clustering of similar words. In *Proceedings of the 17th international conference on Computational linguistics-Volume 2*, pages 768–774. Association for Computational Linguistics.

Ling, W., Luís, T., Marujo, L., Astudillo, R. F., Amir, S., Dyer, C., Black, A. W., and Trancoso, I. (2015). Finding function in form: Compositional character models for open vocabulary word representation. In *Proceedings of Empirical Methods for Natural Language Processing (EMNLP)*.

Liu, B. (2015). *Sentiment Analysis: Mining Opinions, Sentiments, and Emotions*. Cambridge University Press.

Liu, D. C. and Nocedal, J. (1989). On the limited memory BFGS method for large scale optimization. *Mathematical programming*, 45(1-3):503–528.

Manning, C. D., Raghavan, P., Schütze, H., et al. (2008). *Introduction to information retrieval*, volume 1. Cambridge university press.

Manning, C. D. and Schütze, H. (1999). *Foundations of Statistical Natural Language Processing*. MIT press, Cambridge, Massachusetts.

Marcus, M. P., Marcinkiewicz, M. A., and Santorini, B. (1993). Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330.

Martins, A. F. T., Smith, N. A., and Xing, E. P. (2009). Concise integer linear programming formulations for dependency parsing. In *Proceedings of the Association for Computational Linguistics (ACL)*, pages 342–350, Suntec, Singapore.

Martins, A. F. T., Smith, N. A., Xing, E. P., Aguiar, P. M. Q., and Figueiredo, M. A. T. (2010). Turbo parsers: Dependency parsing by approximate variational inference. In *Proceedings of Empirical Methods for Natural Language Processing (EMNLP)*, pages 34–44.

McClosky, D., Charniak, E., and Johnson, M. (2006). Effective self-training for parsing. In *Proceedings of the main conference on human language technology conference of the North American Chapter of the Association of Computational Linguistics*, pages 152–159. Association for Computational Linguistics.

McDonald, R., Crammer, K., and Pereira, F. (2005a). Online large-margin training of dependency parsers. In *Proceedings of the Association for Computational Linguistics (ACL)*, pages 91–98, Ann Arbor, Michigan.

McDonald, R., Hannan, K., Neylon, T., Wells, M., and Reynar, J. (2007). Structured models for fine-to-coarse sentiment analysis. In *Proceedings of ACL*.

McDonald, R., Pereira, F., Ribarov, K., and Hajič, J. (2005b). Non-projective dependency parsing using spanning tree algorithms. In *Proceedings of Empirical Methods for Natural Language Processing (EMNLP)*, pages 523–530.

McLendon, S. (2003). Evidentials in eastern pomo with a comparative survey of the category in other pomoan languages. *TYPOLOGICAL STUDIES IN LANGUAGE*, 54:101–130.

Meyers, A., Reeves, R., Macleod, C., Szekely, R., Zielinska, V., Young, B., and Grishman, R. (2004). The nombank project: An interim report. In *HLT-NAACL 2004 workshop: Frontiers in corpus annotation*, pages 24–31.

Mihalcea, R., Chklovski, T. A., and Kilgarriff, A. (2004). The senseval-3 english lexical sample task. In *Proceedings of SENSEVAL-3*, pages 25–28, Barcelona, Spain. Association for Computational Linguistics.

Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013a). Efficient estimation of word representations in vector space. In *Proceedings of International Conference on Learning Representations*.

Mikolov, T., Deoras, A., Povey, D., Burget, L., and Cernocky, J. (2011). Strategies for training large scale neural network language models. In *Automatic Speech Recognition and Understanding (ASRU), 2011 IEEE Workshop on*, pages 196–201. IEEE.

Mikolov, T., Karafiát, M., Burget, L., Cernockỳ, J., and Khudanpur, S. (2010). Recurrent neural network based language model. In *INTERSPEECH*, pages 1045–1048.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J. (2013b). Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems*, pages 3111–3119.

Mikolov, T., Yih, W.-t., and Zweig, G. (2013c). Linguistic regularities in continuous space word representations. In *Proceedings of the North American Chapter of the Association for Computational Linguistics (NAACL)*, pages 746–751.

Miller, M., Sathi, C., Wiesenthal, D., Leskovec, J., and Potts, C. (2011). Sentiment flow through hyperlink networks. In *Proceedings of the International Conference on Web and Social Media (ICWSM)*.

Miller, S., Guinness, J., and Zamanian, A. (2004). Name tagging with word clusters and discriminative training. In *Proceedings of the North American Chapter of the Association for Computational Linguistics (NAACL)*, pages 337–342, Boston, MA.

Minka, T. P. (1999). From hidden markov models to linear dynamical systems. Tech. Rep. 531, Vision and Modeling Group of Media Lab, MIT.

Minsky, M. and Papert, S. (1969). Perceptrons.

Mintz, M., Bills, S., Snow, R., and Jurafsky, D. (2009). Distant supervision for relation extraction without labeled data. In *Proceedings of the Association for Computational Linguistics (ACL)*, pages 1003–1011, Suntec, Singapore.

Mnih, A. and Teh, Y. W. (2012). A fast and simple algorithm for training neural probabilistic language models. In *Proceedings of the International Conference on Machine Learning (ICML)*.

Mohri, M., Pereira, F., and Riley, M. (2002). Weighted finite-state transducers in speech recognition. *Computer Speech & Language*, 16(1):69–88.

Mohri, M., Rostamizadeh, A., and Talwalkar, A. (2012). *Foundations of machine learning*. MIT press.

Muralidharan, A. and Hearst, M. A. (2013). Supporting exploratory text analysis in literature study. *Literary and linguistic computing*, 28(2):283–295.

Murphy, K. P. (2012). *Machine Learning: A Probabilistic Perspective*. The MIT Press.

Nakagawa, T., Inui, K., and Kurohashi, S. (2010). Dependency tree-based sentiment classification using crfs with hidden variables. In *Proceedings of the North American Chapter of the Association for Computational Linguistics (NAACL)*, pages 786–794, Los Angeles, CA.

Navigli, R. (2009). Word sense disambiguation: A survey. *ACM Computing Surveys (CSUR)*, 41(2):10.

Neal, R. M. and Hinton, G. E. (1998). A view of the em algorithm that justifies incremental, sparse, and other variants. In *Learning in graphical models*, pages 355–368. Springer.

Nemirovski, A. and Yudin, D. (1978). On Cezari's convergence of the steepest descent method for approximating saddle points of convex-concave functions. *Soviet Math. Dokl.*

Neuhaus, P. and Bröker, N. (1997). The complexity of recognition of linguistically adequate dependency grammars. In *eacl*, pages 337–343.

Nigam, K. and Ghani, R. (2000). Analyzing the effectiveness and applicability of co-training. In *Proceedings of the ninth international conference on Information and knowledge management*, pages 86–93. ACM.

Nigam, K., McCallum, A. K., Thrun, S., and Mitchell, T. (2000). Text classification from labeled and unlabeled documents using em. *Machine learning*, 39(2-3):103–134.

Nivre, J. (2004). Incrementality in deterministic dependency parsing. In *Proceedings of the Workshop on Incremental Parsing: Bringing Engineering and Cognition Together*, pages 50–57. Association for Computational Linguistics.

Nivre, J., Hall, J., Nilsson, J., Chanev, A., Eryigit, G., Kübler, S., Marinov, S., and Marsi, E. (2007). Maltparser: A language-independent system for data-driven dependency parsing. *Natural Language Engineering*, 13(2):95–135.

Nivre, J. and Nilsson, J. (2005). Pseudo-projective dependency parsing. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, pages 99–106. Association for Computational Linguistics.

Novikoff, A. B. (1962). On convergence proofs on perceptrons. In *Proceedings of the Symposium on the Mathematical Theory of Automata*, volume 12, pages 615–622.

Och, F. J. and Ney, H. (2003). A systematic comparison of various statistical alignment models. *Computational linguistics*, 29(1):19–51.

Owoputi, O., OConnor, B., Dyer, C., Gimpel, K., and Schneider, N. (2012). Part-of-speech tagging for twitter: Word clusters and other advances. Technical Report CMU-ML-12-107, Carnegie Mellon University.

Pak, A. and Paroubek, P. (2010). Twitter as a corpus for sentiment analysis and opinion mining. In *LREC*, volume 10, pages 1320–1326.

Palmer, M., Gildea, D., and Kingsbury, P. (2005). The proposition bank: An annotated corpus of semantic roles. *Computational linguistics*, 31(1):71–106.

Pang, B. and Lee, L. (2005). Seeing stars: Exploiting class relationships for sentiment categorization with respect to rating scales. In *Proceedings of the Association for Computational Linguistics (ACL)*, pages 115–124, Ann Arbor, Michigan.

Pang, B. and Lee, L. (2008). Opinion mining and sentiment analysis. *Foundations and trends in information retrieval*, 2(1-2):1–135.

Pang, B., Lee, L., and Vaithyanathan, S. (2002). Thumbs up?: sentiment classification using machine learning techniques. In *Proceedings of Empirical Methods for Natural Language Processing (EMNLP)*, pages 79–86.

Pantel, P. and Lin, D. (2002). Discovering word senses from text. In *KDD*, pages 613–619. ACM.

Pennington, J., Socher, R., and Manning, C. (2014). Glove: Global vectors for word representation. In *Proceedings of Empirical Methods for Natural Language Processing (EMNLP)*, pages 1532–1543.

Pereira, F. (2000). Formal grammar and information theory: together again? *Philosophical Transactions of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, 358(1769):1239–1253.

Pereira, F., Tishby, N., and Lee, L. (1993). Distributional clustering of english words. In *Proceedings of the 31st annual meeting on Association for Computational Linguistics*, pages 183–190. Association for Computational Linguistics.

Petrov, S., Barrett, L., Thibaux, R., and Klein, D. (2006). Learning accurate, compact, and interpretable tree annotation. In *Proceedings of the Association for Computational Linguistics (ACL)*.

Petrov, S., Das, D., and McDonald, R. (2012). A universal part-of-speech tagset. In *Proceedings of LREC*.

Petrov, S. and Klein, D. (2007). Improved inference for unlexicalized parsing. In *Proceedings of the North American Chapter of the Association for Computational Linguistics (NAACL)*, pages 404–411.

Popel, M., Marecek, D., Stepánek, J., Zeman, D., and Zabokrtskỳ, Z. (2013). Coordination structures in dependency treebanks. In *Proceedings of the Association for Computational Linguistics (ACL)*, pages 517–527, Sophia, Bulgaria.

Punyakanok, V., Roth, D., and Yih, W.-t. (2008). The importance of syntactic parsing and inference in semantic role labeling. *Computational Linguistics*, 34(2):257–287.

Rabiner, L. (1989). A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286.

Raghunathan, K., Lee, H., Rangarajan, S., Chambers, N., Surdeanu, M., Jurafsky, D., and Manning, C. (2010). A multi-pass sieve for coreference resolution. In *Proceedings of Empirical Methods for Natural Language Processing (EMNLP)*, pages 492–501.

Rao, D. and Ravichandran, D. (2009). Semi-supervised polarity lexicon induction. In *Proceedings of the European Chapter of the Association for Computational Linguistics (EACL)*, pages 675–682.

Ratinov, L. and Roth, D. (2009). Design challenges and misconceptions in named entity recognition. In *Proceedings of the Thirteenth Conference on Computational Natural Language Learning*, pages 147–155. Association for Computational Linguistics.

Ratnaparkhi, A., Reynar, J., and Roukos, S. (1994). A maximum entropy model for prepositional phrase attachment. In *Proceedings of the workshop on Human Language Technology*, pages 250–255. Association for Computational Linguistics.

Read, J. (2005). Using emoticons to reduce dependency in machine learning techniques for sentiment classification. In *Proceedings of the ACL student research workshop*, pages 43–48. Association for Computational Linguistics.

Reisinger, J. and Mooney, R. J. (2010). Multi-prototype vector-space models of word meaning. In *Proceedings of the North American Chapter of the Association for Computational Linguistics (NAACL)*, pages 109–117, Los Angeles, CA.

Resnik, P. and Smith, N. A. (2003). The web as a parallel corpus. *Computational Linguistics*, 29(3):349–380.

Riloff, E. (1996). Automatically generating extraction patterns from untagged text. In *Proceedings of the national conference on artificial intelligence*, pages 1044–1049.

Roark, B., Saraclar, M., and Collins, M. (2007). Discriminative¡ i¿ n¡/i¿-gram language modeling. *Computer Speech & Language*, 21(2):373–392.

Robert, C. and Casella, G. (2013). *Monte Carlo statistical methods*. Springer Science & Business Media.

Rosenfeld, R. (1996). A maximum entropy approach to adaptive statistical language modelling. *Computer Speech & Language*, 10(3):187–228.

Rush, A. M. and Petrov, S. (2012). Vine pruning for efficient multi-pass dependency parsing. In *Proceedings of the North American Chapter of the Association for Computational Linguistics (NAACL)*, pages 498–507.

Sato, M.-A. and Ishii, S. (2000). On-line em algorithm for the normalized gaussian network. *Neural computation*, 12(2):407–432.

Schmid, H. (1994). Probabilistic part-of-speech tagging using decision trees. In *Proceedings of the international conference on new methods in language processing*, volume 12, pages 44–49. Manchester, UK.

Settles, B. (2010). Active learning literature survey. *University of Wisconsin, Madison*, 52(55-66):11.

Shalev-Shwartz, S., Singer, Y., and Srebro, N. (2007). Pegasos: Primal estimated sub-GrAdient SOlver for SVM. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 807–814.

Shen, D. and Lapata, M. (2007). Using semantic roles to improve question answering. In *Proceedings of Empirical Methods for Natural Language Processing (EMNLP)*, pages 12–21.

Shi, L. and Mihalcea, R. (2004). An algorithm for open text semantic parsing. In *Proceedings of the 3rd Workshop on RObust Methods in Analysis of Natural Language Data*, pages 59–67. Association for Computational Linguistics.

Shieber, S. M. (1985). Evidence against the context-freeness of natural language. *Linguistics and Philosophy*, 8(3):333–343.

Sipser, M. (2012). *Introduction to the Theory of Computation*. Cengage Learning.

Smith, D. A. and Eisner, J. (2008). Dependency parsing by belief propagation. In *Proceedings of Empirical Methods for Natural Language Processing (EMNLP)*, pages 145–156, Honolulu, HI.

Smith, D. A. and Smith, N. A. (2007). Probabilistic models of nonprojective dependency trees. In *Proceedings of Empirical Methods for Natural Language Processing (EMNLP)*, pages 132–140.

Smith, N. A. (2011). Linguistic structure prediction. *Synthesis Lectures on Human Language Technologies*, 4(2):1–274.

Smith, N. J. and Levy, R. (2013). The effect of word predictability on reading time is logarithmic. *Cognition*, 128(3):302–319.

Snow, R., O'Connor, B., Jurafsky, D., and Ng, A. Y. (2008). Cheap and fast—but is it good?: evaluating non-expert annotations for natural language tasks. In *Proceedings of Empirical Methods for Natural Language Processing (EMNLP)*, pages 254–263, Honolulu, HI.

Socher, R., Bauer, J., Manning, C. D., and Ng, A. Y. (2013a). Parsing with compositional vector grammars. In *Proceedings of the Association for Computational Linguistics (ACL)*, Sophia, Bulgaria.

Socher, R., Perelygin, A., Wu, J. Y., Chuang, J., Manning, C. D., Ng, A. Y., and Potts, C. (2013b). Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of Empirical Methods for Natural Language Processing (EMNLP)*.

Song, L., Boots, B., Siddiqi, S. M., Gordon, G. J., and Smola, A. J. (2010). Hilbert space embeddings of hidden markov models. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 991–998.

Soon, W. M., Ng, H. T., and Lim, D. C. Y. (2001). A machine learning approach to coreference resolution of noun phrases. *Computational linguistics*, 27(4):521–544.

Spitkovsky, V. I., Alshawi, H., Jurafsky, D., and Manning, C. D. (2010). Viterbi training improves unsupervised dependency parsing. In *CONLL*, pages 9–17.

Sproat, R., Black, A., Chen, S., Kumar, S., Ostendorf, M., and Richards, C. (2001). Normalization of non-standard words. *Computer Speech & Language*, 15(3):287–333.

Sra, S., Nowozin, S., and Wright, S. J. (2012). *Optimization for machine learning*. MIT Press.

Sundermeyer, M., Schlüter, R., and Ney, H. (2012). Lstm neural networks for language modeling. In *INTERSPEECH*.

Surdeanu, M., Màrquez, L., Carreras, X., and Comas, P. R. (2007). Combination strategies for semantic role labeling. *Journal of Artificial Intelligence Research*, pages 105–151.

Tarjan, R. E. (1977). Finding optimum branchings. *Networks*, 7(1):25–35.

Taskar, B., Guestrin, C., and Koller, D. (2003). Max-margin markov networks. In *Neural Information Processing Systems (NIPS)*.

Tausczik, Y. R. and Pennebaker, J. W. (2010). The psychological meaning of words: LIWC and computerized text analysis methods. *Journal of Language and Social Psychology*, 29(1):24–54.

Teh, Y. W. (2006). A hierarchical bayesian language model based on pitman-yor processes. In *Proceedings of the Association for Computational Linguistics (ACL)*, pages 985–992.

Tesnière, L. (1966). *Éléments de syntaxe structurale*. Klincksieck, Paris, second edition.

Thomas, M., Pang, B., and Lee, L. (2006). Get out the vote: Determining support or opposition from Congressional floor-debate transcripts. In *Proceedings of Empirical Methods for Natural Language Processing (EMNLP)*, pages 327–335.

Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 267–288.

Tsochantaridis, I., Hofmann, T., Joachims, T., and Altun, Y. (2004). Support vector machine learning for interdependent and structured output spaces. In *Proceedings of the twenty-first international conference on Machine learning*, page 104. ACM.

Turney, P. (2002). Thumbs up or thumbs down? semantic orientation applied to unsupervised classification of reviews. In *Proceedings of the Association for Computational Linguistics (ACL)*, pages 417–424.

Van Gael, J., Vlachos, A., and Ghahramani, Z. (2009). The infinite hmm for unsupervised pos tagging. In *Proceedings of Empirical Methods for Natural Language Processing (EMNLP)*, pages 678–687, Singapore.

Vaswani, A., Zhao, Y., Fossum, V., and Chiang, D. (2013). Decoding with large-scale neural language models improves translation. In *Proceedings of Empirical Methods for Natural Language Processing (EMNLP)*, pages 1387–1392.

Viterbi, A. (1967). Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE transactions on Information Theory*, 13(2):260–269.

Wainwright, M. J. and Jordan, M. I. (2008). Graphical models, exponential families, and variational inference. *Foundations and Trends® in Machine Learning*, 1(1-2):1–305.

Weaver, W. (1955). Translation. *Machine translation of languages*, 14:15–23.

Wei, G. C. and Tanner, M. A. (1990). A monte carlo implementation of the em algorithm and the poor man's data augmentation algorithms. *Journal of the American Statistical Association*, 85(411):699–704.

Wilson, T., Wiebe, J., and Hoffmann, P. (2005). Recognizing contextual polarity in phrase-level sentiment analysis. In *Proceedings of Empirical Methods for Natural Language Processing (EMNLP)*, pages 347–354.

Wu, B. Y. and Chao, K.-M. (2004). *Spanning trees and optimization problems*. CRC Press.

Xu, W., Liu, X., and Gong, Y. (2003). Document clustering based on non-negative matrix factorization. In *SIGIR*, pages 267–273. ACM.

Yang, Y. and Eisenstein, J. (2015). Unsupervised multi-domain adaptation with feature embeddings. In *Proceedings of the North American Chapter of the Association for Computational Linguistics (NAACL)*, Denver, CO.

Yarowsky, D. (1995). Unsupervised word sense disambiguation rivaling supervised methods. In *Proceedings of the Association for Computational Linguistics (ACL)*, pages 189–196. Association for Computational Linguistics.

Zaidan, O. F. and Callison-Burch, C. (2011). Crowdsourcing translation: Professional quality from non-professionals. In *Proceedings of the Association for Computational Linguistics (ACL)*, pages 1220–1229, Portland, OR.

Zettlemoyer, L. S. and Collins, M. (2005). Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *Proceedings of UAI*.

Zhang, T. (2004). Solving large scale linear prediction problems using stochastic gradient descent algorithms. In *Proceedings of the twenty-first international conference on Machine learning*, page 116. ACM.

Zhang, Y., Lei, T., Barzilay, R., Jaakkola, T., and Globerson, A. (2014). Steps to excellence: Simple inference with refined scoring of dependency trees. In *Proceedings of the Association for Computational Linguistics (ACL)*, pages 197–207, Baltimore, MD.

Zhu, X. and Ghahramani, Z. (2002). Learning from labeled and unlabeled data with label propagation. Technical report, Technical Report CMU-CALD-02-107, Carnegie Mellon University.