A Technical Introduction to Statistical Natural Language Processing

Jacob Eisenstein

July 12, 2017

Contents

Contents **1**

I Words, bags of words, and features **9**

1 Linear classification and features **11**
- 1.1 Review of basic probability 14
- 1.2 Naïve Bayes 21

2 Discriminative learning **31**
- 2.1 Perceptron 32
- 2.2 Loss functions and large margin classification 34
- 2.3 Logistic regression 39
- 2.4 Optimization 43
- 2.5 *Additional topics in classification 45
- 2.6 Summary of learning algorithms 48
- 2.7 Non-linear classification 49

3 Linguistic applications of classification **53**
- 3.1 Sentiment and opinion analysis 53
- 3.2 Word sense disambiguation 57
- 3.3 Design decisions for text classification 60
- 3.4 Evaluating classifiers 64
- 3.5 Building datasets 71

4 Learning without supervision **77**
- 4.1 K-means clustering 78
- 4.2 The Expectation Maximization (EM) Algorithm 79
- 4.3 Applications of EM 84
- 4.4 *Other approaches to learning with latent variables 87

II Sequences and trees **91**

5 Language models **93**
5.1 N-gram language models . . . 94
5.2 Smoothing and discounting . . . 97
5.3 Recurrent neural network language models . . . 102
5.4 Evaluating language models . . . 108
5.5 Out-of-vocabulary words . . . 110

6 Sequence labeling **113**
6.1 Sequence labeling as classification . . . 113
6.2 Sequence labeling as structure prediction . . . 115
6.3 The Viterbi algorithm . . . 117
6.4 Hidden Markov Models . . . 121
6.5 Discriminative sequence labeling . . . 128
6.6 *Unsupervised sequence labeling . . . 139

7 Applications of sequence labeling **143**
7.1 Part-of-speech tagging . . . 143
7.2 Tokenization . . . 152
7.3 Shallow parsing . . . 152
7.4 Named entity recognition . . . 152
7.5 Dialogue acts . . . 154
7.6 Code switching . . . 154

8 Finite-state automata **155**
8.1 Automata and languages . . . 156
8.2 Weighted Finite State Automata . . . 161
8.3 Semirings . . . 165
8.4 Finite state transducers . . . 167
8.5 Weighted FSTs . . . 169
8.6 Applications of finite state composition . . . 171
8.7 Discriminative structure prediction . . . 173

9 Context-free grammars **175**
9.1 Is English a regular language? . . . 175
9.2 Context-Free Languages . . . 177
9.3 Constituents . . . 180
9.4 A simple grammar of English . . . 182
9.5 Grammar equivalence and normal form . . . 188

10 Context-free Parsing **191**

10.1 Deterministic bottom-up parsing 191
10.2 Ambiguity in parsing 194
10.3 Weighted Context-Free Grammars 198
10.4 Improving Parsing by Refined Non-terminals 203
10.5 Discriminative parsing 212

11 Dependency Parsing 215
11.1 Dependency grammar 215
11.2 Graph-based dependency parsing 220
11.3 Transition-based dependency parsing 227
11.4 Applications 230

III Meaning 233

12 Compositional semantics 235
12.1 Meaning and denotation 236
12.2 Logical representations of meaning 236
12.3 Semantic parsing and the lambda calculus 241
12.4 Learning semantic parsers 247

13 Predicate-argument semantics 255
13.1 Semantic roles 257
13.2 Semantic role labeling 261
13.3 Abstract Meaning Representation 270
13.4 Applications of Predicate-Argument Semantics 274

14 Distributional and distributed semantics 281
14.1 The distributional hypothesis 281
14.2 Design decisions for word representations 283
14.3 Distributional semantics 285
14.4 Distributed representations 291

15 Reference Resolution 297
15.1 Forms of referring expressions 298
15.2 Learning for coreference resolution 304
15.3 Entity linking and multi-document coreference resolution 308

16 Discourse 311
16.1 Discourse relations in the Penn Discourse Treebank 311
16.2 Rhetorical Structure Theory 311
16.3 Centering 311

16.4 Lexical cohesion and text segmentation 311
16.5 Dialogue 311

IV Applications 313

17 Information extraction 315
17.1 Entities 316
17.2 Relations 318
17.3 Events and processes 318
17.4 Facts, beliefs, and hypotheticals 318

18 Machine translation 319
18.1 Statistical machine translation in the noisy channel model 320
18.2 Neural machine translation 328
18.3 Decoding 332
18.4 *Syntactic MT 332

Bibliography 337

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, July 12, 2017

Notation

w_m	word token at position m
$\boldsymbol{x}^{(i)}$	a (column) vector of feature counts for instance i, often word counts
$\boldsymbol{x}_{i:j}$	elements i through j (inclusive) of a vector $\boldsymbol{x}$
N	number of training instances
M	length of a sequence (of words or tags)
$\|\mathcal{V}\|$	number of words in vocabulary
$y^{(i)}$	the label for instance i
$\hat{y}$	a predicted label
$\boldsymbol{y}$	a vector of labels
$\mathcal{Y}$	the set of all possible labels
K	number of possible labels $K = \|\mathcal{Y}\|$
$\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)})$	feature vector for instance i with label $y^{(i)}$
$\boldsymbol{\theta}$	a (column) vector of weights
$\Pr(A)$	probability of event A
$\mathrm{p}_B(b)$	the marginal probability of random variable B (often implicit) taking value b
$\mathcal{Y}(\boldsymbol{w})$	the set of possible tag sequences for the word sequence $\boldsymbol{w}$
$\Diamond$	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 $\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y)$. 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

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

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 $Y =$ spam should be larger than the weight for the offset feature for $Y =$ 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

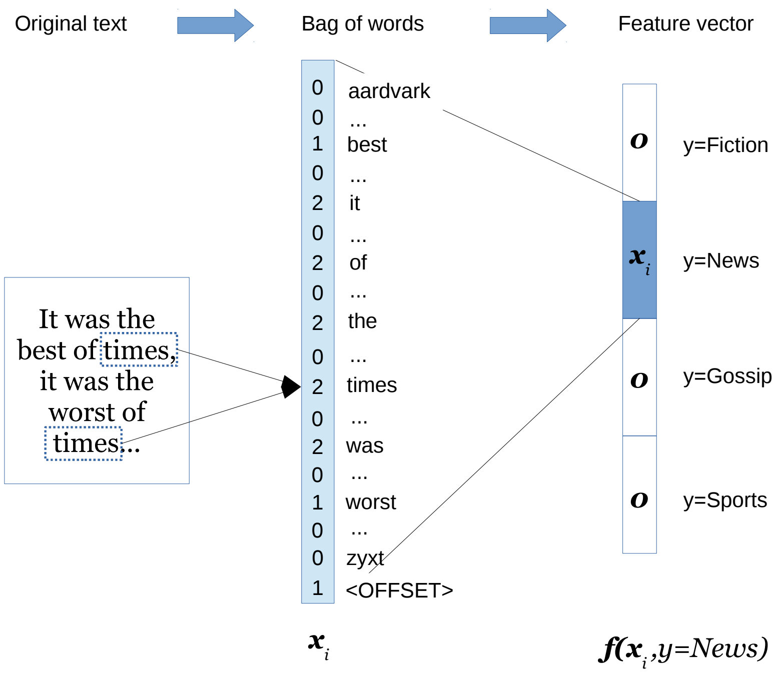


Figure 1.1: The bag-of-words and feature vector representations, for a hypothetical text classification task.

appears in the associated dictionary. For example,[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 labels to automatically set the weights for a classifier. An important tool for this is probability.

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

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,

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

1.1.1 Probabilities of event combinations

Because events are **sets** of outcomes, we can use set theoretic operations such as 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:

- 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, $\{TT, HH\}$.

[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`.

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,

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

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,

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

[3] [todo: add figure]

1.1.2 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:

$$\Pr(T \cap G) = \Pr(T \mid G) \times \Pr(G) = 0.95 \times 10^{-5} \tag{1.22}$$

$$\Pr(T \cap \neg G) = \Pr(T \mid \neg G) \times \Pr(\neg G) = 0.005 \times (1 - 10^{-5}) \approx 0.005 \tag{1.23}$$

$$\Pr(T) = \Pr(T \cap G) + \Pr(T \cap \neg G) \tag{1.24}$$

$$= 0.95 \times 10^{-5} + 0.005 \approx 0.005. \tag{1.25}$$

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

$$\Pr(G \mid T) = \frac{\Pr(T \mid G) \Pr(G)}{\Pr(T)} \tag{1.26}$$

$$= \frac{0.95 \times 10^{-5}}{0.95 \times 10^{-5} + 0.005 \times (1 - 10^{-5})} \tag{1.27}$$

$$\approx 0.002. \tag{1.28}$$

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.

1.1.3 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:

$$\Pr(\{HT, HH\}) = \Pr(HT) + \Pr(HH) = \frac{1}{4} + \frac{1}{4} = \frac{1}{2} \tag{1.29}$$

$$\Pr(\{HH, TH\}) = \Pr(HH) + \Pr(TH) = \frac{1}{4} + \frac{1}{4} = \frac{1}{2} \tag{1.30}$$

$$\Pr(\{HT, HH\}) \times \Pr(\{HH, TH\}) = \frac{1}{2} \times \frac{1}{2} = \frac{1}{4} \tag{1.31}$$

$$\Pr(\{HT, HH\} \cap \{HH, TH\}) = \Pr(HH) = \frac{1}{4} \tag{1.32}$$

$$= \Pr(\{HT, HH\}) \times \Pr(\{HH, TH\}). \tag{1.33}$$

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 $A \perp B \mid C$.

1.1.4 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 Y 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 $\Pr(Y = 1)$ is the sum of the probabilities of these outcomes, $\Pr(Y = 1) = \frac{1}{4} + \frac{1}{4} + \frac{1}{4} = \frac{3}{4}$.
- 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 $\text{p}_X(x)$, 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 $\int_x \text{p}_X(x)dx = 1$ and $\forall x, \text{p}_X(x) \geq 0$.

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

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

1.1.5 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 has a special offer for cold days: they give a 1% discount on poutine 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.)

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

1.1.6 Modeling and estimation

Probabilistic models give us a principled way to reason about random events and random variables, and to make predictions about the future. Let's consider the coin toss example. We can model each toss as a random event, with probability θ of the event H, and probability $1 - \theta$ of the complementary event T. If we write a random variable X as the total number of heads on three coin flips, then the distribution of X depends on θ. In this case, X is distributed as a **binomial random variable**, meaning that it is drawn from a binomial distribution, with **parameters** $(\theta, N = 3)$. We write:

$$X \sim \text{Binomial}(\theta, N = 3). \tag{1.36}$$

This is a probabilistic model of X. The binomial distribution has a number of known properties that enable us to make statements about the X, such as its expected value, the likelihood that its value will fall within some interval, etc.

Now suppose that θ is unknown, but we have run an experiment, in which we executed N trials, and obtained x heads. We can **estimate** θ by the principle of **maximum likelihood**:

$$\hat{\theta} = \underset{\theta}{\text{argmax}}\, \text{p}_X(x; \theta, N). \tag{1.37}$$

This says that our estimate $\hat{\theta}$ should be the value that maximizes the likelihood of the data we have observed. The semicolon indicates that θ and N are parameters of the probability function. The likelihood $\text{p}_X(x; \theta, N)$ can be computed from the binomial distribution,

$$\text{p}_X(x; \theta, N) = \frac{N!}{x!(N-x)!} \theta^x (1-\theta)^{N-x}. \tag{1.38}$$

This likelihood is proportional to the product of the probability of individual outcomes: for example, the sequence T, H, H, T, H would have probability $\theta^2(1-\theta)^3$. The term $\frac{N!}{x!(N-x)!}$ arises from the many possible orderings by which we could obtain x heads on N trials. This term is constant in θ, so it can be ignored.

We can maximize likelihood by taking the derivative and setting it equal to zero. In practice, we usually maximize log-likelihood, which is a monotonic function of the likelihood, and is easier to manipulate mathematically.

$$\ell(\theta) = x \log \theta + (N - x) \log(1 - \theta) \tag{1.39}$$

$$\frac{\partial \ell(\theta)}{\partial \theta} = \frac{x}{\theta} - \frac{N - x}{1 - \theta} \tag{1.40}$$

$$\frac{N - x}{1 - \theta} = \frac{x}{\theta} \tag{1.41}$$

$$\frac{N - x}{x} = \frac{1 - \theta}{\theta} \tag{1.42}$$

$$\frac{N}{x} - 1 = \frac{1}{\theta} - 1 \tag{1.43}$$

$$\hat{\theta} = \frac{x}{N}. \tag{1.44}$$

In this case, the maximum likelihood estimate is equal to $\frac{x}{N}$, the fraction of trials that came up heads. This intuitive solution is also known as the **relative frequency estimate**, since it is equal to the relative frequency of the outcome.

Is maximum likelihood estimation always the right choice? Suppose you conduct one trial, and get heads — would you conclude that $\theta = 1$, so this coin is guaranteed to give heads? If not, then you must have some **prior expectation** about θ. To incorporate this prior information, we can treat θ as a random variable, and use Bayes rule:

$$\mathrm{p}(\theta \mid x; N) = \frac{\mathrm{p}(x \mid \theta) \times \mathrm{p}(\theta)}{\mathrm{p}(x)} \tag{1.45}$$

$$\propto \mathrm{p}(x \mid \theta) \times \mathrm{p}(\theta) \tag{1.46}$$

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}}\, \mathrm{p}(x \mid \theta) \times \mathrm{p}(\theta). \tag{1.47}$$

This it the **maximum a posteriori** (MAP) estimate. Given a form for $\mathrm{p}(\theta)$, you can derive the MAP estimate using the same approach that was used to derive the maximum likelihood estimate.

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 construcst the weights $\boldsymbol{\theta}$ and the feature function $\boldsymbol{f}(\boldsymbol{x}, y)$ so that the inner product $\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y)$ is equal to the joint log-probability $\log \mathrm{p}(\boldsymbol{x}, y)$. We can then set the weights to maximize the probability of a labeled dataset, $\{\boldsymbol{x}^{(i)}, y^{(i)}\}_{i \in 1 \dots N}$, where each tuple $\langle \boldsymbol{x}^{(i)}, y^{(i)} \rangle$ is a labeled instance.

To carry out this strategy, We first need to define the probability $\mathrm{p}(\{\boldsymbol{x}^{(i)}, y^{(i)}\}_{i\in 1\ldots N})$. We will do that through a **generative model**, which describes a hypothesized stochastic process that has generated the observed data.[5]

- 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)**.

$$\mathrm{p}(\{\boldsymbol{x}^{(i)}, y^{(i)}\}_{i\in 1\ldots N}; \boldsymbol{\mu}, \boldsymbol{\phi}) = \prod_{i=1}^{N} \mathrm{p}(\boldsymbol{x}^{(i)}, y^{(i)}; \boldsymbol{\mu}, \boldsymbol{\phi}) \tag{1.48}$$

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 describes the how $\boldsymbol{x}^{(i)}$ is sampled, conditional on $y^{(i)}$. It invokes the **multinomial distribution**, which is a probability distribution over vectors of non-negative counts. The probability mass function for this distribution is:

$$\mathrm{p}_{\text{mult}}(\boldsymbol{x}; \boldsymbol{\phi}) = B(\boldsymbol{x}) \prod_{j}^{V} \phi_j^{x_j} \tag{1.49}$$

$$B(\boldsymbol{x}) = \frac{\left(\sum_j^V x_j\right)!}{\prod_j^V (x_j!)} \tag{1.50}$$

As in the categorical distribution, the parameter ϕ_j can be interpreted as a probability: specifically, the probability that any given token in the document is the word j. The multinomial distribution involves a product over words, with each term in the product

[5]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).

equal to the probability of the word ϕ_j exponentiated by the count x_j. Words that have zero count play no role in this product, because $\phi_j^0 = 1$ for all ϕ_j.

The term $B(\boldsymbol{x})$ doesn't depend on $\boldsymbol{\phi}$, and can usually be ignored. Can you see why we need this term at all?[6] We will return to this issue shortly.

We can write $\mathrm{p}(\boldsymbol{x} \mid y; \boldsymbol{\phi})$ to indicate the conditional probability of word counts $\boldsymbol{x}$ given label y, with parameter $\boldsymbol{\phi}$, which is equal to $\mathrm{p}_{\text{mult}}(\boldsymbol{x}; \boldsymbol{\phi}_y)$. By specifying the multinomial distribution for $\mathrm{p}_{\boldsymbol{x}|y}$, 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.[7] We'll see this more clearly later, when we show how MNB is an example of linear classification.

1.2.1 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,

$$\mathrm{p}_W(\textit{multinomial}) \times \mathrm{p}_W(\textit{Naive}) \times \mathrm{p}_W(\textit{Bayes}). \tag{1.51}$$

Formally, this is a model for the joint probability of the word *sequence* $\boldsymbol{w}$ and the label y, $\mathrm{p}(\boldsymbol{w}, y)$, not the joint probability of the word *counts* $\boldsymbol{x}$ and the label y, $\mathrm{p}(\boldsymbol{x}, y)$.

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, $B(\boldsymbol{x})$. This means that the probability for a vector of counts $\boldsymbol{x}$ is larger 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 of 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 $B(\boldsymbol{x}) = \frac{(\sum_j x_j)!}{\prod_j x_j!}$.

[6]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.

[7]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.

From the perspective of classification, none of this matters, because it has nothing to do with the label y or the parameters $\boldsymbol{\phi}$ and $\boldsymbol{\mu}$. 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 $\boldsymbol{\phi}$ (which are defined below).

1.2.2 Prediction

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

$$\hat{y} = \operatorname*{argmax}_y \mathrm{p}(\boldsymbol{x}, y; \boldsymbol{\mu}, \boldsymbol{\phi}) \tag{1.52}$$

$$= \operatorname*{argmax}_y \mathrm{p}(\boldsymbol{x} \mid y; \boldsymbol{\phi}) \mathrm{p}(y; \boldsymbol{\mu}) \tag{1.53}$$

$$= \operatorname*{argmax}_y \log \mathrm{p}(\boldsymbol{x} \mid y; \boldsymbol{\phi}) + \log \mathrm{p}(y; \boldsymbol{\mu}) \tag{1.54}$$

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.

$$\log \mathrm{p}(\boldsymbol{x} \mid y; \boldsymbol{\phi}) + \log \mathrm{p}(y; \boldsymbol{\mu}) = \log \left[B(\boldsymbol{x}) \prod_j^V \boldsymbol{\phi}_{y,j}^{x_j} \right] + \log \mu_y \tag{1.55}$$

$$= \log B(\boldsymbol{x}) + \sum_j^V x_j \log \phi_{y,j} + \log \mu_y \tag{1.56}$$

$$= \log B(\boldsymbol{x}) + \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y), \tag{1.57}$$

where

$$\boldsymbol{\theta} = [\boldsymbol{\theta}^{(1)\top}, \boldsymbol{\theta}^{(2)\top}, \ldots, \boldsymbol{\theta}^{(K)\top}]^\top \tag{1.58}$$

$$\boldsymbol{\theta}^{(y)} = [\log \phi_{y,1},\ \log \phi_{y,2},\ \ldots,\ \log \phi_{y,V},\ \log \mu_y]^\top \tag{1.59}$$

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 $\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y)$ 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.

1.2.3 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

$$\phi_{y,j} = \frac{\sum_{i:y^{(i)}=y} x_{i,j}}{\sum_{j'}^{V} \sum_{i:y^{(i)}=y} x_{i,j'}} = \frac{\text{count}(y,j)}{\sum_{j'}^{V} \text{count}(y,j')} \tag{1.60}$$

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

Our prediction rule in Equation 1.52 is to choose $\hat{y}$ to maximize the joint probability $\text{p}(\boldsymbol{x}, y)$. 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**: a set of annotated examples for which we observe both the text and the true label, $\{\boldsymbol{x}^{(i)}, y^{(i)}\}_{i\in 1\dots N}$. Based on the generative model that we have defined, the log-likelihood is:

$$L(\boldsymbol{\phi}, \boldsymbol{\mu}) = \sum_{i}^{N} \log \text{p}_{\text{mult}}(\boldsymbol{x}_i; \boldsymbol{\phi}_{y^{(i)}}) + \log \text{p}_{\text{cat}}(y^{(i)}; \boldsymbol{\mu}). \tag{1.61}$$

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

$$\begin{aligned} L(\boldsymbol{\phi}) =& \sum_{i}^{N} \log \text{p}_{\text{mult}}(\boldsymbol{x}^{(i)}; \boldsymbol{\phi}_{y^{(i)}}) & (1.62)\\ =& \sum_{i}^{N} \log B(\boldsymbol{x}) \prod_{j}^{V} \phi_{y^{(i)},j}^{x_{i,j}} & (1.63)\\ =& \sum_{i}^{N} \log B(\boldsymbol{x}) + \sum_{j}^{V} x_{i,j} \log \phi_{y^{(i)},j}, & (1.64) \end{aligned}$$

where $B(\boldsymbol{x})$ is 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:

$$\forall y, \sum_{j=1}^{V} \phi_{y,j} = 1 \tag{1.65}$$

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

$$\ell[\boldsymbol{\phi}_y] = \sum_{i:Y^{(i)}=y} \sum_{j}^{V} x_{ij} \log \phi_{y,j} - \lambda(\sum_{j}^{V} \phi_{y,j} - 1) \tag{1.66}$$

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

$$0 = \sum_{i:Y^{(i)}=y} x_{i,j}/\phi_{y,j} - \lambda \tag{1.67}$$

$$\lambda \phi_{y,j} = \sum_{i:Y^{(i)}=y} x_{i,j} \tag{1.68}$$

$$\phi_{y,j} \propto \sum_{i:Y^{(i)}=y} x_{i,j} = \sum_i \delta(Y^{(i)} = y) x_{i,j}, \tag{1.69}$$

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,

$$\phi_{y,j} = \frac{\sum_{i:Y^{(i)}=y} x_{i,j}}{\sum_{j'=1}^{V} \sum_{i:Y^{(i)}=y} x_{i,j'}} \tag{1.70}$$

$$= \frac{\text{count}(y,j)}{\sum_{j'=1}^{V} \text{count}(y,j')}. \tag{1.71}$$

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.

1.2.4 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.

$$\phi_{y,j} = \frac{\alpha + \sum_{i:Y^{(i)}=y} x_j^{(i)}}{\sum_{j'=1}^{V} \left(\alpha + \sum_{i:Y^{(i)}=y} x_{i,j'}\right)} = \frac{\alpha + \text{count}(y,j)}{V\alpha + \sum_{j'=1}^{V} \text{count}(y,j')} \tag{1.72}$$

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**.[8]

- 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 $\alpha \to \infty$, 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 can just go for low bias.

1.2.5 The Naïvety of Naïve Bayes

Naïve Bayes is simple to work with: estimation and prediction can be done in closed form, and the nice probabilistic interpretation makes it relatively easy to extend the model. 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:

$$\mathrm{p}(\textit{naïve}, \textit{Bayes} \mid y) = \mathrm{p}(\textit{naïve} \mid y) \times \mathrm{p}(\textit{Bayes} \mid y). \tag{1.73}$$

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

$$\mathrm{p}(\textit{naïve Bayes}) > \mathrm{p}(\textit{naïve}) \times \mathrm{p}(\textit{Bayes}) \tag{1.74}$$

  or...

$$\mathrm{p}(\textit{naïve Bayes}) < \mathrm{p}(\textit{naïve}) \times \mathrm{p}(\textit{Bayes}). \tag{1.75}$$

  Apply the chain rule!

[8]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.

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.

1.2.6 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**.

Exercises

[todo: make exercises]

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,

$$
\begin{aligned}
\log \mathrm{p}(y^{(i)}, \boldsymbol{x}^{(i)}) = & \log \mathrm{p}(\boldsymbol{x}^{(i)} \mid y^{(i)}) + \log \mathrm{p}(y^{(i)}) && (2.1)\\
= & \sum_j \log \mathrm{p}(x_{i,j} \mid y^{(i)}) + \log \mathrm{p}(y^{(i)}) && (2.2)\\
= & \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) && (2.3)
\end{aligned}
$$

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. Bag-of-words features violate the assumption of conditional independence — for example, the probability that a document will contain the word *naïve* is surely higher given that it also contains the word *Bayes* — but this 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**.

Many of these "rich" features violate the Naïve Bayes independence assumption more severely. Consider what happens if we add feature capturing the word prefix. Under the Naïve Bayes assumption, we make the following approximation:

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

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

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

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 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**:

[1]I have been deliberately vague about the stopping criterion; this is discussed later in the chapter.

[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. [todo: keep this?]

Algorithm 1 Perceptron learning algorithm

```
1: procedure PERCEPTRON($\boldsymbol{x}^{(1:N)}, y^{(1:N)}$)
2:     $t \leftarrow 0$
3:     $\boldsymbol{\theta}_t \leftarrow \mathbf{0}$
4:     repeat
5:         $t \leftarrow t + 1$
6:         Select an instance $i$
7:         $\hat{y} \leftarrow \text{argmax}_y \, \boldsymbol{\theta} \cdot_{t-1} \boldsymbol{f}(\boldsymbol{x}^{(i)}, y)$
8:         if $\hat{y} \neq y^{(i)}$ then
9:             $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} + \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \boldsymbol{f}(\boldsymbol{x}^{(i)}, \hat{y})$
10:        else
11:            $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1}$
12:    until tired
13:    return $\boldsymbol{\theta}$
```

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, $\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y^{(i)})$, is at least ρ greater than inner product of $\boldsymbol{\theta}$ and the feature function for every other possible label, $\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y')$.*

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

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

[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).

2.1.1 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. The algorithm sketch indicates that we compute the average by keeping a running sum, $\boldsymbol{m} \leftarrow \boldsymbol{m} + \boldsymbol{\theta}$. However, this is inefficient, because it requires $|\boldsymbol{\theta}|$ operations to update the running sum. In NLP problems, $\boldsymbol{f}(\boldsymbol{x}, y)$ is typically sparse, so $|\boldsymbol{\theta}| \gg |\boldsymbol{f}(\boldsymbol{x}, y)|$ for any individual $(\boldsymbol{x}, y)$. This means that the computation of the running sum will be much more expensive than the computation of the update to $\boldsymbol{\theta}$ itself, which requires only $2 \times |\boldsymbol{f}(\boldsymbol{x}, y)|$ operations. One of the exercises is to sketch a more efficient algorithm for computing the averaged weights.

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**, and was introduced in chapter 1). When the accuracy on the heldout data starts to decrease, the learning algorithm has begun to **overfit** the training set. At this point, it is probably best to stop; this stopping criterion is known as **early stopping**.

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 $\mathrm{p}(\{\boldsymbol{x}^{(i)}, y^{(i)}\}_{i \in 1 \ldots N})$. 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

Algorithm 2 Averaged perceptron learning algorithm

```
1:  procedure AVG-PERCEPTRON($\boldsymbol{x}^{(1:N)}, y^{(1:N)}$)
2:      $t \leftarrow 0$
3:      $\boldsymbol{\theta}_0 \leftarrow 0$
4:      repeat
5:          $t \leftarrow t + 1$
6:          Select an instance $i$
7:          $\hat{y} \leftarrow \text{argmax}_y \, \boldsymbol{\theta} \cdot_{t-1} \boldsymbol{f}(\boldsymbol{x}^{(i)}, y)$
8:          if $\hat{y} \neq y^{(i)}$ then
9:              $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} + \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \boldsymbol{f}(\boldsymbol{x}^{(i)}, \hat{y})$
10:         else
11:             $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1}$
12:         $\boldsymbol{m} \leftarrow \boldsymbol{m} + \boldsymbol{\theta}_t$
13:     until tired
14:     $\overline{\boldsymbol{\theta}} \leftarrow \frac{1}{t}\boldsymbol{m}$
15:     return $\overline{\boldsymbol{\theta}}$
```

can therefore be viewed as a **loss function**,

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

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

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

This minimization problem is identical to the maximum-likelihood estimation problem that we solved in the previous chapter. Framing it as minimization may seem backwards, but loss functions provide a very general framework in which to compare many approaches to machine learning. For example, an alternative loss function is the **zero-one loss**,

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

This loss function is closely related to accuracy, which may seem ideal. But it is **non-convex**[4] and discontinuous, which means that it is combinatorially difficult to optimize.

[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).

The perceptron optimizes the following loss function:

$$\ell_{\text{perceptron}}(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) = \max_{y} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y) - \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}), \tag{2.13}$$

which is a **hinge loss** with the hinge point at zero. When $\hat{y} = y^{(i)}$, the loss is zero; otherwise, it increases linearly with the gap between the score for the predicted label $\hat{y}$ and the score for the true label $y^{(i)}$. To see why this is the loss function optimized by the perceptron, just take the derivative with respect to $\boldsymbol{\theta}$,

$$\frac{\partial}{\partial \boldsymbol{\theta}} \ell_{\text{perceptron}}(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) = \boldsymbol{f}(\boldsymbol{x}^{(i)}, \hat{y}) - \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}). \tag{2.14}$$

One way to minimize our loss is to take a step of magnitude τ in the opposite direction of this gradient,

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \tau \frac{\partial}{\partial \boldsymbol{\theta}} \ell_{\text{perceptron}}(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) = \qquad \boldsymbol{\theta} + \tau(\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \boldsymbol{f}(\boldsymbol{x}^{(i)}, \hat{y})). \tag{2.15}$$

When the step size $\tau = 1$, this is identical to the perceptron update.

This loss function has some pros and cons with respect to the joint likelihood loss implied by Naïve Bayes.

- Both ℓ_{NB} and $\ell_{\text{perceptron}}$ are convex, making them relatively easy to optimize. However, ℓ_{NB} can be optimized in closed form, while $\ell_{\text{perceptron}}$ requires iterating over the dataset multiple times.
- 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 correct answers equally. Even if $\boldsymbol{\theta}$ only gives the correct answer by a tiny margin, the loss is still zero.

This last comment suggests a potential problem. Suppose a test example is very close to a training example, but not identical. If the classifier only gets the correct answer on the training example by a small margin, then it may get the test instance wrong. To formalize this intuition, let's define the **margin** as

$$\gamma(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) = \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \max_{y \neq y^{(i)}} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y) \tag{2.16}$$

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 **margin loss**,

$$\ell_{\text{margin}}(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) = \begin{cases} 0, & \gamma(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) \geq 1, \\ 1 - \gamma(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}), & \text{otherwise} \end{cases} \tag{2.17}$$

Equivalently, we can write $\ell_{\text{margin}}(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) = \left(1 - \gamma(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)})\right)_+$, where $(x)_+ = \max(0, x)$. The margin loss is zero if we have a margin of at least 1 between the score for the true label and the best-scoring alternative, which we have written $\hat{y}$. It is equivalent to the hinge loss defined above, but shifted to the right on the x-axis. The margin and zero-one loss functions are shown in Figure 2.1. Note that the margin loss is a convex upper bound on the zero-one loss.

2.2.1 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 loss. To see this, let us redefine the margin as,

$$\gamma(\boldsymbol{\theta}, \boldsymbol{x}^{(i)}, y^{(i)}) = \min_{y \neq y^{(i)}} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y) \tag{2.18}$$

$$= \min_{y \neq y^{(i)}} s\boldsymbol{u} \cdot \Big(\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \boldsymbol{f}(\boldsymbol{x}^{(i)}, y)\Big). \tag{2.19}$$

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 $\forall s \geq s^*, \gamma(s\boldsymbol{u}^*, \boldsymbol{x}^{(i)}, y^{(i)}) \geq 1$. Given many possible $\boldsymbol{\theta}$ that obtain zero margin loss, we may prefer 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,[6] which, in its most basic form, solves the following optimization problem,

$$\begin{aligned} \min_{\boldsymbol{\theta}}. \quad & ||\boldsymbol{\theta}||_2^2 \\ s.t. \quad & \forall_i \ell_{\text{margin}}(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) = 0. \end{aligned} \tag{2.20}$$

In realistic settings, we do not know whether there is any feasible solution — that is, whether there exists any $\boldsymbol{\theta}$ so that the margin 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 loss be exactly zero, we require that it be

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

[6]Instances near the margin are called **support vectors**. In some optimization methods for this model, the support vectors play an especially important role, motivating the name.

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:

$$\begin{aligned} \min_{\boldsymbol{\theta}} \quad & ||\boldsymbol{\theta}||_2^2 + C\sum_i \xi_i \\ s.t. \quad & \forall_i \ell_{\text{margin}}(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) \leq \xi_i \\ & \forall_i \xi_i \geq 0. \end{aligned} \tag{2.21}$$

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 (§ 1.2.4), 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 2.21, we can use Lagrange multipliers to convert it into the unconstrained **primal form**,[7]

$$\min_{\boldsymbol{\theta}} \quad \frac{\lambda}{2}||\boldsymbol{\theta}||_2^2 + \sum_i \ell_{\text{margin}}(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}), \tag{2.22}$$

where λ is a tunable parameter that can be computed from the term C in Equation 2.21. A generic way to minimize such objective functions is **gradient descent**: moving along the gradient (obtained by differentiating with respect to $\boldsymbol{\theta}$), until the gradient is equal to zero.[8]

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

$$L_{SVM} = \frac{\lambda}{2}||\boldsymbol{\theta}||_2^2 + \sum_i^N \ell_{\text{margin}}(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) \tag{2.23}$$

$$= \frac{\lambda}{2}||\boldsymbol{\theta}||_2^2 + \sum_i^N (1 - \gamma(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}))_+ \tag{2.24}$$

$$= \frac{\lambda}{2}||\boldsymbol{\theta}||_2^2 + \sum_i^N (1 - \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) + \max_{y \neq y^{(i)}} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y))_+ \tag{2.25}$$

[7]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.

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

Let us define the **cost** of a misclassification as,

$$c(y^{(i)}, \hat{y}) = \begin{cases} 1, & y^{(i)} \neq \hat{y} \\ 0, & \text{otherwise.} \end{cases} \tag{2.26}$$

We can then simplify Equation 2.25,

$$L_{SVM} = \frac{\lambda}{2}||\boldsymbol{\theta}||_2^2 + \sum_i^N (\max_y(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y) + c(y^{(i)}, y)) - \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}))_+, \tag{2.27}$$

where we now maximize over all $y \in \mathcal{Y}$, favoring labels that are both high-scoring (as measured by $\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y)$) and wrong (as measured by $c(y^{(i)}, y)$). When the highest-scoring such label is $y = y^{(i)}$, then the margin constraint is satisfied, and the loss for this instance is zero.

Then the (sub)gradient of Equation 2.27 is:

$$\frac{\partial L_{SVM}}{\partial \boldsymbol{\theta}} = \lambda \boldsymbol{\theta} + \sum_i^N \boldsymbol{f}(\boldsymbol{x}^{(i)}, \hat{y}) - \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}), \tag{2.28}$$

where $\hat{y} = \text{argmax}_y\, \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y) + c(y^{(i)}, y))$. If we were to update $\boldsymbol{\theta}$ by adding this gradient, this would be very similar to the perceptron algorithm: the only difference is the additional cost term $c(y^{(i)}, y)$, which derives from the margin constraint.

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 and support-vector machines are error-driven algorithms: the learning objective is closely related to the number of errors on the training data, and will be maximized when there are zero errors. Probabilistic and error-driven approaches each 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 $\text{p}(\boldsymbol{x}, y)$.

- We have used the chain rule to factor this joint probability as $\text{p}(\boldsymbol{x}, y) = \text{p}(\boldsymbol{x} \mid y) \times \text{p}(y)$.

- 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 $\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y)$ 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 $\exp(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y))$, 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,

$$\mathrm{p}(y \mid \boldsymbol{x}) = \frac{\exp(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y))}{\sum_{y' \in \mathcal{Y}} \exp(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y'))}. \tag{2.29}$$

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

$$\begin{aligned}
L(\boldsymbol{\theta}) &= \log \mathrm{p}(\boldsymbol{y}^{(1:N)} \mid \boldsymbol{x}^{(1:N)}; \boldsymbol{\theta}) && (2.30)\\
&= \log \prod_{i=1}^{N} \mathrm{p}(y^{(i)} \mid \boldsymbol{x}^{(i)}; \boldsymbol{\theta}) && (2.31)\\
&= \sum_{i=1}^{N} \log \mathrm{p}(y^{(i)} \mid \boldsymbol{x}^{(i)}; \boldsymbol{\theta}) && (2.32)\\
&= \sum_{i=1}^{N} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \log \sum_{y' \in \mathcal{Y}} \exp\left(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y')\right). && (2.33)
\end{aligned}$$

The final line is obtained by plugging in Equation 2.29 and taking the logarithm.[9,10] Inside the sum, we have the (additive inverse of the) **logistic loss**.

- In binary classification, we can write this as

$$\ell_{\text{logistic}}(\boldsymbol{\theta}; \boldsymbol{x}_i, y^{(i)}) = -(y^{(i)} \boldsymbol{\theta}^\top \boldsymbol{x}_i - \log(1 + \exp(\boldsymbol{\theta} \cdot \boldsymbol{x}_i))) \tag{2.34}$$

[9]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.

[10]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.

- In multi-class classification, we have,

$$\ell_{\text{logistic}}(\boldsymbol{\theta}; \boldsymbol{x}_i, y^{(i)}) = -(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \log \sum_{y' \in \mathcal{Y}} \exp(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y'))) \tag{2.35}$$

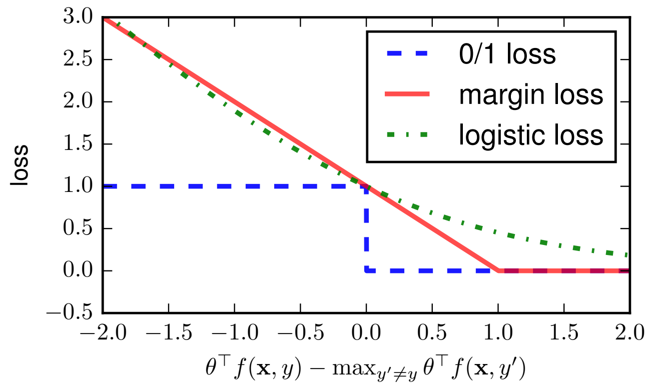


Figure 2.1: Margin, zero-one, and logistic loss functions

The logistic loss is shown in Figure 2.1. 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. This means that the objective function can always be improved by chosing the correct label with more confidence.

2.3.1 Regularization

As with the margin-based algorithms described in § 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,

$$\log N(\theta_j; 0, \sigma^2) \propto -\frac{1}{2\sigma^2}\theta_j^2, \tag{2.36}$$

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 $\boldsymbol{f}(\boldsymbol{x}^{(i)}, y)$.

2.3.2 Gradients

We will optimize $\boldsymbol{\theta}$ through gradient descent. Specific algorithms are described in § 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,

$$\ell(\boldsymbol{\theta}; \boldsymbol{x}_i, y^{(i)}) = -\left(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \log \sum_{y' \in \mathcal{Y}} \exp\left(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y')\right)\right) \tag{2.37}$$

$$\frac{\partial \ell}{\partial \boldsymbol{\theta}} = -\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) + \frac{1}{\sum_{y'' \in \mathcal{Y}} \exp\left(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y'')\right)} \times \sum_{y'} \exp\left(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y')\right) \times \boldsymbol{f}(\boldsymbol{x}^{(i)}, y') \tag{2.38}$$

$$= -\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) + \sum_{y'} \frac{\exp\left(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y')\right)}{\sum_{y'' \in \mathcal{Y}} \exp\left(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y'')\right)} \times \boldsymbol{f}(\boldsymbol{x}^{(i)}, y') \tag{2.39}$$

$$= -\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) + \sum_{y'} \mathrm{p}(y' \mid \boldsymbol{x}^{(i)}; \boldsymbol{\theta}) \times \boldsymbol{f}(\boldsymbol{x}^{(i)}, y') \tag{2.40}$$

$$= -\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) + E_{y|\boldsymbol{x}}[\boldsymbol{f}(\boldsymbol{x}^{(i)}, y)], \tag{2.41}$$

where the final step employs the definition of an expectation (§ 1.1.5). 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 $\mathrm{p}(\boldsymbol{y} \mid \boldsymbol{x})$ is a delta function, $\mathrm{p}(y \mid \boldsymbol{x}) = \delta(y = \hat{y})$, 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:

$$L = \frac{\lambda}{2}||\boldsymbol{\theta}||_2^2 - \sum_{i=1}^{N} \left(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \log \sum_{y' \in \mathcal{Y}} \exp \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y')\right) \tag{2.42}$$

$$\frac{\partial L}{\partial \boldsymbol{\theta}} = \lambda\boldsymbol{\theta} - \sum_{i=1}^{N} \left(\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - E_{y|\boldsymbol{x}}[\boldsymbol{f}(\boldsymbol{x}^{(i)}, y)]\right) \tag{2.43}$$

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,

$$L_{LR} = \frac{\lambda}{2}||\boldsymbol{\theta}||_2^2 - \sum_i \left(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \log \sum_y \exp \left(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y) \right) \right) \tag{2.44}$$

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

$$L_{SVM} = \frac{\lambda}{2}||\boldsymbol{\theta}||_2^2 + \sum_i^N (\max_y (\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y) + c(y^{(i)}, y)) - \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}))_+, \tag{2.45}$$

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.

2.4.1 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,

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t - \eta_t \frac{\partial L}{\partial \boldsymbol{\theta}}, \tag{2.46}$$

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,

$$H_{i,j} = \frac{\partial^2}{\partial w_i \partial w_j} L. \tag{2.47}$$

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).[11] 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.

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

$$\begin{aligned}
\boldsymbol{\theta}^{(t+1)} \leftarrow & \boldsymbol{\theta}^{(t)} - \eta_t \frac{\partial L_{LR}}{\partial \boldsymbol{\theta}} && (2.48)\\
= & \boldsymbol{\theta}^{(t)} - \eta_t \left(\lambda \boldsymbol{\theta}^{(t)} - \sum_i^N \left(\boldsymbol{f}(\boldsymbol{x}_i, y^{(i)}) - E_{y|\boldsymbol{x}}[\boldsymbol{f}(\boldsymbol{x}_i, y)] \right) \right) && (2.49)\\
= & (1 - \lambda \eta_t) \boldsymbol{\theta}^{(t)} + \eta_t \left(\sum_i^N \boldsymbol{f}(\boldsymbol{x}_i, y^{(i)}) - E_{y|\boldsymbol{x}}[\boldsymbol{f}(\boldsymbol{x}_i, y)] \right) && (2.50)\\
\approx & (1 - \lambda \eta_t) \boldsymbol{\theta}^{(t)} + N \eta_t \left(\boldsymbol{f}(\boldsymbol{x}_{i(t)}, y_{i(t)}) - E_{y|\boldsymbol{x}}[\boldsymbol{f}(\boldsymbol{x}_{i(t)}, y)] \right) && (2.51)
\end{aligned}$$

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 2.28.

Algorithm 3 Stochastic gradient descent for logistic regression

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

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

$$E_{y|\boldsymbol{x}}[\boldsymbol{f}(\boldsymbol{x}^{(i)}, y)] = \sum_{y' \in \mathcal{Y}} \mathrm{p}(y' \mid \boldsymbol{x}^{(i)}; \boldsymbol{\theta}) \boldsymbol{f}(\boldsymbol{x}^{(i)}, y'). \tag{2.52}$$

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

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).

2.4.3 *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:

$$\boldsymbol{g}_t = \lambda\boldsymbol{\theta} - \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) + \sum_{y' \in \mathcal{Y}} \mathrm{p}(y' \mid \boldsymbol{x}^{(i)})\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) \tag{2.53}$$

$$\theta_j^{(t+1)} \leftarrow \theta_j^{(t)} - \frac{\eta}{\sqrt{\sum_{t'=1}^{t} g_{t,j}^2}} g_{t,j}, \tag{2.54}$$

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

2.5.1 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:

$$\begin{aligned} \min w. \quad & \frac{1}{2}||\boldsymbol{\theta} - \boldsymbol{\theta}_t||^2 + C\xi_t \\ s.t. \quad & \ell_{\text{hinge}}(\boldsymbol{\theta}; \boldsymbol{x}_i, y^{(i)}) \leq \xi_t, \xi_t \geq 0 \end{aligned} \tag{2.55}$$

By forming another Lagrangian, it is possible to show that the solution to Equation 2.55 is,

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \tau_t(\boldsymbol{f}(y^{(i)}, \boldsymbol{x}^{(i)}) - \boldsymbol{f}(\hat{y}, \boldsymbol{x}^{(i)})) \tag{2.56}$$

$$\tau_t = \min\left(C, \frac{\ell(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)})}{||\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \boldsymbol{f}(\boldsymbol{x}^{(i)}, \hat{y})||^2}\right), \tag{2.57}$$

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.[12] 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.

2.5.2 Other regularizers

In Equation 2.42, 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 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

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

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.

2.5.3 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 $\boldsymbol{\theta} \cdot \boldsymbol{x}$ 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 $\mathrm{p}(y \mid \boldsymbol{x})$.
- 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 $\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y)$.

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:

$$\forall j, \sum_{i=1}^{N} f_j(\boldsymbol{x}^{(i)}, y^{(i)}) = \sum_{i=1}^{N} \sum_{y \in \mathcal{Y}} \mathrm{p}(y \mid \boldsymbol{x}^{(i)}; \boldsymbol{\theta}) f_j(\boldsymbol{x}^{(i)}, y) \tag{2.58}$$

Note that this constraint will be met exactly when the derivative of the likelihood function (Equation 2.41) 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,

$$H(\mathrm{p}_{y|\boldsymbol{x}}) = -\sum_{\boldsymbol{x}\in\mathcal{X}} \mathrm{p}_{\boldsymbol{x}}(\boldsymbol{x}) \sum_{y\in\mathcal{Y}} \mathrm{p}_{y|\boldsymbol{x}}(y \mid \boldsymbol{x}) \log \mathrm{p}_{y|\boldsymbol{x}}(y \mid \boldsymbol{x}), \tag{2.59}$$

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,

$$\tilde{H}(\mathrm{p}_{y|\boldsymbol{x}}) = -\frac{1}{N}\sum_{i}\sum_{y\in\mathcal{Y}} \mathrm{p}_{y|\boldsymbol{x}}(y \mid \boldsymbol{x}^{(i)}) \log \mathrm{p}_{y|\boldsymbol{x}}(y \mid \boldsymbol{x}^{(i)}). \tag{2.60}$$

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 2.58. 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; lack of margin can lead to overfitting.

Support vector machine *Pros*: optimizes an error-based metric, usually resulting in high accuracy; overfitting is controlled by a regularization parameter. *Cons*: 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, SVM, and logistic regression. In non-probabilistic settings, I usually reach for averaged perceptron first if I am coding from scratch. If probabilities are necessary, I use logistic regression.

2.7 Non-linear classification

The feature spaces that we consider in NLP are usually quite large: at least on the order of the size of the vocabulary, often much more. When the number of (unique) features is larger than the number of instances, it is possible to learn a linear classifier that perfectly classifies any training data – even when the labels are chosen at random. This problem is intensified for non-linear classification, where the space of possible separators is increased.[13]

2.7.1 Feature expansion, decision trees, and boosting

The simplest 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 $\boldsymbol{x} \odot \boldsymbol{x}$ and the cross-product $\boldsymbol{x} \otimes \boldsymbol{x}$.

Decision tree and **boosting** algorithms (Freund et al., 1999) learn non-linear conjunctions of features. For example, a decision tree algorithm might learn a classification rule that chooses a class only when a combination of three features is active. Decision trees are typically grown recursively: a feature is chosen to divide the data into two or more subsets, with the goal of maximizing the homogeneity of labels in each subset. Then additional features are chosen for each subset so that homogeneity is increased. Boosting algorithms combine aspects of decision trees and feature expansion: they iteratively build classifiers that are weighted sums of many "short" decision trees, sometimes called **decision stumps** Mohri et al. (2012).

Although decision tree boosting is widely successful in machine learning competitions (Chen and Guestrin, 2016), it is rarely used in contemporary natural language processing. There are two main reasons for this: first, linear classifiers are still very effective for many NLP task; second, deep neural networks have become the dominant approach for non-linear classification in language processing tasks.

[13]The set of possible linear classifiers is infinite. **VC-dimension** provides a theory for comparing the expressiveness of various classifiers (Mohri et al., 2012).

2.7.2 Neural networks

The core idea behind **deep learning** is perform non-linear classification by passing the inputs through a series of non-linear transformations. Each of these transformations is learned in a supervised fashion, by **backpropagating** from a loss function. For document classification, convolutional neural networks are a popular approach, because of their ability to capture multi-word units. Surveys are offered by Goldberg (2015) and Cho (2015).

Convolutional neural networks

Recurrent neural networks

[todo: or maybe save these for language modeling chapter?]

2.7.3 Kernels

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 methods are often used in combination with the support vector machine, which has a **dual form** in which kernel functions can be inserted in place of inner products on the feature vectors.

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 17.

	Naive Bayes	Logistic Regression	Perceptron	SVM
Objective	Joint likelihood	Conditional likelihood	Hinge loss	Margin 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	$\theta_{ij} = \frac{c(x_i, y=j)+\alpha}{c(y=j)+\lvert\mathcal{V}\rvert\alpha}$	$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}} = \sum_i \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - E[\boldsymbol{f}(\boldsymbol{x}^{(i)}, y)]$	$\boldsymbol{\theta}^{(t)} \leftarrow \boldsymbol{\theta}^{(t-1)} + \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \boldsymbol{f}(\boldsymbol{x}^{(i)}, \hat{y})$	$\boldsymbol{\theta}^{(t)} \leftarrow \boldsymbol{\theta}^{(t-1)} + \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \boldsymbol{f}(\boldsymbol{x}^{(i)}, \hat{y})$
tuning	smoothing α	regularizer $\lambda \lVert\boldsymbol{\theta}\rVert_2^2$	weight averaging	slack penalty C, or regularizer λ
complexity	$\mathcal{O}(N\lvert\mathcal{V}\rvert)$	$\mathcal{O}(NT\lvert\mathcal{V}\rvert)$	$\mathcal{O}(NT\lvert\mathcal{V}\rvert)$	$\mathcal{O}(NT\lvert\mathcal{V}\rvert)$
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, $|\mathcal{V}|$ = number of features, T = number of training iterations.

Exercises

1. As noted in the discussion of averaged perceptron in § 2.1.1, the computation of the running sum $\boldsymbol{m} \leftarrow \boldsymbol{m} + \boldsymbol{\theta}$ is unnecessarily expensive, requiring $K \times |\mathcal{V}|$ operations. Give an alternative way to compute the averaged weights $\overline{\boldsymbol{\theta}}$, with complexity that is independent of $|\mathcal{V}|$ and linear in the sum of feature sizes $\sum_{i=1}^{N} |\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)})|$.

2. [todo: reconcile with notation in this chapter] Suppose you have two datasets $D_1 = \{x_i^{(1)}, y_i^{(1)}\}_{i \in 1 \ldots N_1}$ and $D_2 = \{x_j^{(2)}, y_j^{(2)}\}_{j \in 1 \ldots N_2}$, with each $y \in \{-1, 1\}$, and each $x \in \mathbb{R}^P$.

   - Let $w^{(1)}$ be the unregularized logistic regression (LR) coefficients from training on dataset D_1, under the model, $P(y \mid x; w) = \sigma(y(x \cdot w))$, with σ indicating the sigmoid function and $x \cdot w$ indicating the dot product of the features x and the coefficients w.
   - Let $w^{(2)}$ be the unregularized LR coefficients (same model) from training on dataset D_2.
   - Let w^* be the unregularized LR coefficients from training on the combined dataset $D_1 \cup D_2$.

   Under these conditions, prove that for any feature n,

$$
\begin{aligned}
w_n^* &\geq \min(w_n^{(1)}, w_n^{(2)}) \\
w_n^* &\leq \max(w_n^{(1)}, w_n^{(2)}).
\end{aligned}
$$

Chapter 3

Linguistic applications of classification

Having learned some techniques for classification, we will now see how they can be applied to some classical problems in natural language technology. Later in this chapter, we discuss some of the design decisions involved in text classification, as well as evaluation practices.

3.1 Sentiment and opinion analysis

A popular application of text classification is to automatically determine 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).[1]

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).

[1]Comprehensive surveys on sentiment analysis and related problems are offered by Pang and Lee (2008) and Liu (2015).

- 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).

The bag-of-words model is a good fit for sentiment analysis at the document level: if the document is long enough, we would expect the words associated with its true sentiment to overwhelm the others. Indeed, **lexicon-based sentiment analysis** avoids machine learning altogether, and classifies documents by counting words against positive and negative sentiment word lists (Taboada et al., 2011).

The problem becomes more tricky for short documents, such as single-sentence reviews or social media posts. In these documents, linguistic issues like **negation** and **irrealis** (Polanyi and Zaenen, 2006) — events that are hypothetical or otherwise non-factual — can make bag-of-words classification ineffective. Consider the following examples:

(3.1) *That's not bad for the first day.*

(3.2) *This is not the worst thing that can happen.*

(3.3) *It would be nice if you acted like you understood.*

(3.4) *There is no reason at all to believe that the polluters are suddenly going to become reasonable.* (Wilson et al., 2005)

(3.5) *This film should be brilliant. The actors are first grade. Stallone plays a happy, wonderful man. His sweet wife is beautiful and adores him. He has a fascinating gift for living life fully. It sounds like a great plot,* ***however****, the film is a failure.* (Pang et al., 2002)

A minimal solution is to move from a bag-of-words model to a bag-of-**bigrams** model, where each base feature is a pair of adjacent words, e.g.,

$$\langle \textit{that's}, \textit{not} \rangle, \langle \textit{not}, \textit{bad} \rangle, \langle \textit{bad}, \textit{for} \rangle, \ldots \tag{3.1}$$

Bigrams can handle relatively straightforward cases, such as when an adjective is immediately negated; trigrams would be required to extend to larger contexts (e.g., *not the worst*). But it should be clear that this approach will not scale to the more complex examples, such as (3.4) and (3.5). More sophisticated solutions try to account for the syntactic structure of the sentence (Wilson et al., 2005; Socher et al., 2013b), or apply more complex classifiers such as **convolutional neural networks** (Kim, 2014), which are described in § 2.7.

3.1.1 Related problems

Subjectivity Closely related to sentiment analysis is **subjectivity detection**, which requires identifying the parts of a text that express subjective opinions, as well as other non-factual content such speculation and hypotheticals (Riloff and Wiebe, 2003). This can be done by treating each sentence as a separate document, and then applying a bag-of-words classifier: indeed, Pang and Lee (2004) do exactly this, using a training set consisting of (mostly) subjective sentences gathered from movie reviews, and (mostly) objective sentences gathered from plot descriptions. They augment this bag-of-words model with a graph-based algorithm that encourages nearby sentences to have the same subjectivity label.

Stance classification In debates, each participant takes a side: for example, advocating for or against adopting a vegetarian lifestyle or mandating free college education. The problem of stance classification involves identifying an author's position from the text of the argument. In some cases, there is training data available for each position, so that standard document classification techniques can be employed. In other cases, it suffices to classify each document as whether it is in support or opposition of the argument advanced by a previous document (Anand et al., 2011). In the most challenging case, there is no labeled data for any of the stances, so the only possibility is group documents that advocate the same position (Somasundaran and Wiebe, 2009). This is a form of **unsupervised learning**, and will be discussed in chapter 4.

Targeted sentiment analysis The expression of sentiment is often more nuanced than a simple binary label. Consider the following examples:

(3.6) *The vodka was good, but the meat was rotten.*

(3.7) *Go to Heaven for the climate, Hell for the company.* (Mark Twain)

These statements display a mixed overall sentiment: positive towards some entities (e.g., *the vodka*), negative towards others (e.g., *the meat*). **Targeted sentiment analysis** seeks to identify the writer's sentiment towards specific entities (Jiang et al., 2011). This requires identifying the entities in the text and linking them to specific sentiment words — much more than we can do with the classification-based approaches discussed thus far. For example, Kim and Hovy (2006) analyze sentence-internal structure to determine the topic of each sentiment expression.

Aspect-based opinion mining seeks to identify the sentiment of the author of a review towards predefined aspects such as PRICE and SERVICE, or, in the case of (3.7), CLIMATE and COMPANY (Hu and Liu, 2004). If the aspects are not defined in advance, it may again be necessary to employ **unsupervised learning** methods to identify them (e.g., Branavan et al., 2009).

Emotion classification While sentiment analysis is framed in terms of positive and negative categories, psychologists generally regard **emotion** as more multifaceted. For example, Ekman (1992) argues that there are six basic emotions — happiness, surprise, fear, sadness, anger, and contempt — and that they are universal across human cultures. Alm et al. (2005) build a linear classifier for recognizing the emotions expressed in children's stories. The ultimate goal of this work was to improve text-to-speech synthesis, so that stories could be read with intonation that reflected the emotional content. They used bag-of-words features, as well as features capturing the story type (e.g., jokes, folktales), and structural features that reflect the position of each sentence in the story. The task is difficult: even human annotators frequently disagreed with each other, and the best classifiers achieved accuracy between 60-70%.

3.1.2 Alternative approaches to sentiment analysis

Regression A more challenging version of sentiment analysis is to determine not just the class of a document, 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 $\hat{y} = \boldsymbol{\theta} \cdot \boldsymbol{x} + b$, where b is an offset. This approach is called **linear regression**, and sometimes **least squares**, because the regression coefficients $\boldsymbol{\theta}$ are determined by minimizing the squared error, $(y - \hat{y})^2$. If the weights are regularized using a penalty $\lambda||\boldsymbol{\theta}||_2^2$, then the name **ridge regression** is sometimes applied. Unlike logistic regression, both linear regression and ridge regression can be solved in closed form as a system of linear equations.

Ranking In many problems, the labels are ordered but discrete: for example, product reviews are often integers on a scale of $1 - 5$, and grades are on a scale of $A - F$. Such **ranking** problems can be solved by discretizing the score $\boldsymbol{\theta} \cdot \boldsymbol{x}$ into ranks,

$$\hat{y} = \underset{r:\, \boldsymbol{\theta} \cdot \boldsymbol{x} \geq b_r}{\operatorname{argmin}} r, \tag{3.2}$$

where $\boldsymbol{b} = [b_1 = -\infty, b_2, b_3, \ldots, b_K]$ is a vector of boundaries. Crammer and Singer (2001) show that it is possible to learn both the weights and boundaries simultaneously, using a perceptron-like algorithm.

Lexicon-based classification Sentiment analysis is one of the only NLP tasks where hand-crafted feature weights are still widely employed. In **lexicon-based classification** (Taboada et al., 2011), the user creates a list of words for each label, and then classifies each document based on how many of the words from each list are present. In our linear classification framework, this is equivalent to choosing the following weights:

$$\theta_{y,j} = \begin{cases} 1, & j \in \mathcal{L}_y \\ 0, & \text{otherwise,} \end{cases} \tag{3.3}$$

where $\mathcal{L}_y$ is the lexicon for label y. Compared to the machine learning classifiers discussed in the previous chapters, lexicon-based classification may seem primitive. However, supervised machine learning relies on large annotated datasets, which are time-consuming and expensive to produce. If the goal is to distinguish two or more categories in a new domain, it may be simpler to start by writing down a list of words for each category.

An early lexicon was the *General Inquirer* (Stone, 1966). Today, popular sentiment lexicons include `sentiwordnet` (Esuli and Sebastiani, 2006) and an evolving set of lexicons from Liu (2015). For emotions and more fine-grained analysis, *Linguistic Inquiry and Word Count* (LIWC) provides a set of lexicons (Tausczik and Pennebaker, 2010). The MPQA lexicon indicates the polarity of some 8221 terms, as well as whether they are strongly or weakly subjective (Wiebe et al., 2005). A comprehensive comparison of sentiment lexicons is offered by Ribeiro et al. (2016). Given an initial **seed lexicon**, it is possible to automatically expand the lexicon by looking for words that frequently co-occur with words in the seed set (Hatzivassiloglou and McKeown, 1997; Qiu et al., 2011).

3.2 Word sense disambiguation

Consider the the following headlines:

(3.8) *Iraqi head seeks arms*

(3.9) *Prostitutes appeal to Pope*

(3.10) *Drunk gets nine years in violin case*[2]

These headlines 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. Word sense disambiguation is part of a larger field of research called **lexical semantics**, which is concerned with meanings of the words.

At a basic level, the problem of word sense disambiguation is to identify the correct sense for each word token in a document. Part-of-speech ambiguity (e.g., noun versus verb) is usually considered to be a different problem, to be solved at an earlier stage. From a linguistic perspective, senses are not really properties of words, but of **lemmas**, which are canonical forms that stand in for a set of inflected words. For example, *arm*/N is a lemma that includes the inflected form *arms*/N — the /N indicates that it we are referring to the noun form of the word. Similarly, *arm*/V is a lemma that includes the inflected verbs $(arm/\text{V}, arms/\text{V}, armed/\text{V}, arming/\text{V})$. Therefore, word sense disambiguation requires first identifying the correct part-of-speech and lemma for each token, and then choosing the correct sense from the inventory associated with the corresponding lemma.[3]

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

[3]Navigli (2009) provides a survey of approaches for word-sense disambiguation.

3.2.1 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*
- [CONTRIBUTE TO]: *His evasive replies only served to heighten suspicion*
- [PROVIDE]: *We serve only the rawest fish*
- [ENLIST]: *She served in an elite combat unit*
- [JAIL]: *He served six years for a crime he didn't commit*
- [LEGAL]: *They were served with subpoenas*[4]

These sense distinctions are annotated in WORDNET (`http://wordnet.princeton.edu`), a lexical semantic database for English. WordNet consists of roughly 100,000 **synsets**, which are groups of lemmas (or phrases) that are synonymous. An example synset is $\{$*chump*^{1},*fool*^{2}, *sucker*^{1}, *mark*$^9\}$, where the superscripts index the sense of each lemma that is included in the synset: for example, there are at least eight other senses of *mark* that have different meanings, and are not part of this synset. A lemma is **polysemous** if it participates in multiple synsets.

WordNet plays an key role in setting the parameters of the word sense disambiguation problem, and in formalizing lexical semantic knowledge of English. (WordNets have been created for a few dozen other languages, at varying levels of detail.) 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), people agree on roughly 70% of examples using WordNet senses; far better than chance, but perhaps less than we might like.

***Other lexical semantic relations** Besides **synonymy**, WordNet also describes many other lexical semantic 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**;
- **meronymy**: x is a part of y, e.g., WHEEL-BICYCLE; the inverse relationship is **holonymy**.

[4]Several of the examples are adapted from WordNet (Fellbaum, 2010)

Classification of these relations relations can be performed by searching for characteristic patterns between pairs of words, e.g., *X*, <u>*such as*</u> *Y*, which signals hyponymy (Hearst, 1992), or *X* <u>*but*</u> *Y*, which signals antonymy (Hatzivassiloglou and McKeown, 1997). Another approach is to analyze each term's **distributional statistics** (the frequency of its neighboring words). Such approaches are described in detail in chapter 14.

3.2.2 Word sense disambiguation as classification

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

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

(3.12) *The endangered plants play an important role in the local ecosystem.*

It is possible to build a feature vector using the bag-of-words representation, by treating each context as a pseudo-document. We can then construct a feature function for each potential sense y,

$$f(\langle \textit{plant}, \textit{The endangered plants play an} \ldots \rangle, y) = \\ \{\langle \textit{the}, y\rangle : 1, \langle \textit{endangered}, y\rangle : 1, \langle \textit{play}, y\rangle : 1, \langle \textit{an}, y\rangle : 1, \ldots\}$$

As in document classification, many of these features are irrelevant, but a few are very strong indicators. In this example, the context word *endangered* is a strong signal that the intended sense is biology rather than manufacturing. We would therefore expect a learning algorithm to assign high weight to $\langle \textit{endangered}, \textsc{biology}\rangle$, and low weight to $\langle \textit{endangered}, \textsc{manufacturing}\rangle$.[5]

It may also be helpful to go beyond the bag-of-words: for example, one might encode the position of each context word with respect to the target, e.g.,

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

These **collocation features** give more information about the specific role played by each context word. This idea can be taken further by incorporating additional syntactic information about the grammatical role played by each context feature, such as the **dependency path** (see chapter 11).

After deciding on the features, we can train a classifier to predict the sense of each word. A **semantic concordance** is a corpus in which each open-class word (nouns, verbs,

[5]The context bag-of-words can be also used be used to perform word-sense disambiguation without machine learning: the Lesk (1986) algorithm selects the word sense whose dictionary definition best overlaps the local context.

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, annotated as part of the WordNet project (Fellbaum, 2010). SemCor annotations look like this:

(3.13) As of Sunday^1_n night^1_n there was^4_v no word^2_n ...,

with the superscripts indicating the annotated sense of each polysemous word.

As always, supervised classification is only possible if enough labeled examples can be accumulated. This is difficult in word sense disambiguation, because each polysemous lemma requires its own training set: having a good classifier for the senses of *serve* is no help towards disambiguating *plant*. For this reason, **unsupervised** and **semisupervised** methods are particularly important for WSD (e.g., Yarowsky, 1995). These methods will be discussed in chapter 4 and **??**. Unsupervised methods typically lean on the heuristic of "one sense per discourse", which means that a lemma will usually have a single, consistent sense throughout any given document (Gale et al., 1992). Based on this heuristic, we can propagate information from high-confidence instances to lower-confidence instances in the same document (Yarowsky, 1995).

3.3 Design decisions for text classification

Text classification involves a number of design decisions. Some of these decisions, such as smoothing or regularization, are specific to the classifier. These decisions are described in the previous chapter. But even the construction of the feature vector itself involves a number of design decisions, and these decisions can be the most consequential for the classifier's performance.

3.3.1 What is a word?

The bag-of-words representation presupposes that extracting a vector of word counts from text is unambiguous. But text documents are generally represented as a sequences of characters, and the conversion to bag-of-words presupposes a definition of the "words" that are to be counted.

Tokenization

The first subtask for constructing a bag-of-words vector is **tokenization**: converting the text from a sequence of characters to a sequence of **word tokens**. A simple approach is to define a subset of characters as whitespace, and then split the text on these tokens. However, whitespace-based tokenization is not ideal: we may want to split conjunctions like *isn't* and hyphenated phrases like *prize-winning* and *half-asleep*, and we likely want to separate words from commas and periods that immediately follow them. This suggests

Whitespace	Isn't	Ahab	Ahab?	;)					
Treebank	Is	n't	Ahab	,	Ahab	?	;	)	
Tweet	Isn't	Ahab	,	Ahab	?	;)			
TokTok (Dehdari, 2014)	Isn	'	t	Ahab	,	Ahab	?	;	)

Figure 3.1: The output of four `nltk` tokenizers, applied to the string *Isn't Ahab, Ahab? ;)*

that a tokenizer should split on all non-alphanumeric characters, but we would prefer not to split abbreviations like *U.S.* and *Ph.D.* In languages with Roman scripts, tokenization is typically performed using regular expressions, with modules designed to handle each of these cases. For example, the `nltk` package includes a number of tokenizers; the outputs of four of the better-known tokenizers are shown in Figure 3.1. Social media researchers have found that emoticons and other forms of orthographic variation pose new challenges for tokenization, leading to the development of special purpose tokenizers to handle these phenomena (O'Connor et al., 2010).

Tokenization is a language-specific problem, and each language poses unique challenges. For example, Chinese does not include spaces between words, nor any other consistent orthographic markers of word boundaries. A "greedy" approach is to scan the input for character substrings that are in a predefined lexicon. However, Xue et al. (2003) notes that this can be ambiguous, since many character sequences could be segmented in multiple ways. Instead, he trains a classifier to determine whether each Chinese character, or **hanzi**, is a word boundary. More advanced sequence labeling methods for word segmentation are discussed in **??**. Similar problems can occur in languages with alphabetic scripts, such as German, which does not include whitespace in compound nouns, yielding examples such as *Freundschaftsbezeigungen* and *Dilettantenaufdringlichkeiten* [todo: ask German speaker for better examples]. As Twain (1997) argues, "*These things are not words, they are alphabetic processions.*" Social media raises similar problems for English and other languages, with hashtags such as *#TrueLoveInFourWords* requiring decomposition for analysis (Brun and Roux, 2014).

Normalization

After splitting the text into tokens, the next question is which tokens are really distinct. Is it necessary to distinguish *great*, *Great*, and GREAT? Sentence-initial capitalization may be irrelevant to the classification task. The elimination of case distinctions will result in a smaller vocabulary, and thus smaller feature vectors. However, case distinctions might be relevant in some situations: for example, *apple* is a delicious pie filling, while *Apple* is a company dedicated to marketing proprietary dongles and power adapters. For Latin script, case conversion can be performed using unicode string libraries. Many scripts do not have case distinctions (e.g., the Devanagari script used for South Asian languages, the

Original	The	Williams	sisters	are	leaving	this	tennis	centre
Porter stemmer	the	william	sister	are	leav	thi	tenni	centr
Lancaster stemmer	the	william	sist	ar	leav	thi	ten	cent

Figure 3.2: The outputs of the Porter (Porter, 1980) and Lancaster (Paice, 1990) stemmers and the WordNet lemmatizer.

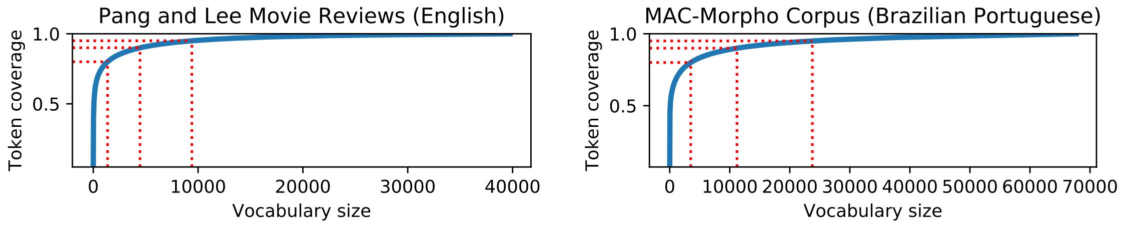
Thai alphabet, and Japanese kana), and case conversion for all scripts may not be available in every programming environment.

Case conversion is an example of **normalization**, which refers to string transformations that remove distinctions that are felt to be irrelevant (Sproat et al., 2001). Other normalizations include the standardization of numbers (e.g., *1,000* to *1000*) and dates (e.g., *August 11, 2015* to *2015/11/08*). Depending on the application, it may even be worthwhile to convert all numbers and dates to special tokens, `!NUM` and `!DATE`. Social media features orthographic phenomena such as expressive lengthening (e.g., *cooooool*), which may also be normalized (Aw et al., 2006; Yang and Eisenstein, 2013). Similarly, historical texts feature spelling variations which may be normalized to a standard form (Baron and Rayson, 2008).

A more extreme form of normalization is to eliminate **inflectional affixes**, such as the *-ed* and *-s* suffixes in English. On this view, *bike*, *bikes*, *biking*, and *biked* all refer to the same underlying concept, so they should be grouped into a single feature. A **stemmer** is a program for eliminating affixes, usually by applying a series of regular expression substitutions. Character-based stemming algorithms are necessarily approximate, as shown in Figure 3.2: the Lancaster stemmer incorrectly identifies *-ers* as an inflectional suffix of *sisters* (by analogy to *fix/fixers*), and both stemmers incorrectly identify *-s* as a suffix of *this* and *Williams*. Fortunately, even inaccurate stemming can improve bag-of-words classification models, by merging related strings and thereby reducing the vocabulary size.

Accurately handling irregular orthography requires word-specific rules. **Lemmatizers** are systems that identify the underlying lemma of a given wordform. They must avoid the over-generalization errors of the stemmers in Figure 3.2, and also handle more complex transformations, such as *geese*$\rightarrow$*goose*. The output of the WordNet lemmatizer is shown in the final line of Figure 3.2. Both stemming and lemmatization are language-specific: an English stemmer or lemmatizer is of little use on a text written in another language. The discipline of **morphology** relates to the study of word-internal structure, and is described in more detail in § 8.1.2.

The value of normalization depends on the data and the task. Normalization reduces the size of the feature space, which can help in generalization. However, there is always the risk of merging away linguistically meaningful distinctions. In supervised machine learning, regularization and smoothing can play a similar role to normalization — preventing the learner from overfitting to rare features — while avoiding the language-



(a) Movie review data in English (b) News articles in Brazilian Portuguese

Figure 3.3: Tradeoff between token coverage (y-axis) and vocabulary size, on the `nltk` movie review dataset, after sorting the vocabulary by decreasing frequency. The red dashed lines indicate 80%, 90%, and 95% coverage.

specific engineering required for accurate normalization. In unsupervised scenarios, such as content-based information retrieval (Manning et al., 2008) and topic modeling (Blei et al., 2003), normalization is more critical.

3.3.2 How many words?

Limiting the size of the feature vector reduces the memory footprint of the resulting models, and increases the speed of prediction. Normalization can help to play this role, but a more direct approach is simply to limit the vocabulary to the N most frequent words in the dataset. For example, in the `movie-reviews` dataset provided with `nltk` (originally from Pang et al., 2002), there are 39,768 word types, and 1.58M tokens. As shown in Figure 3.3a, the most frequent 4000 word types cover 90% of all tokens, offering an order-of-magnitude reduction in the model size. Such ratios are language-specific: in for example, in the Brazilian Portuguese Mac-Morpho corpus (Aluísio et al., 2003), attaining 90% coverage requires more than 10000 word types (Figure 3.3b). This reflects the morphological complexity of Portuguese, which includes many more inflectional suffixes than English.

Eliminating rare words is not always advantageous for classification performance: for example, names, which are typically rare, play a large role in distinguishing topics of news articles. Another way to reduce the size of the feature space is to eliminate **stopwords** such as *the*, *to*, and *and*, which may seem to play little role in expressing the topic, sentiment, or stance. This is typically done by creating a **stoplist** (e.g., `nltk.corpus.stopwords`), and then ignoring all terms that match the list. However, corpus linguists and social psychologists have shown that seemingly inconsequential words can offer surprising insights about the author or nature of the text (Biber, 1991; Chung and Pennebaker, 2007). Furthermore, high-frequency words are unlikely to cause overfitting in well-regularized discriminative classifiers. As with normalization, stopword filtering is more important for unsupervised problems, such as term-based document retrieval; in this case, matching a *to* or *the* in the search query offers little information that the resulting document will meet the goals of the user's search.

Another alternative for controlling model size is **feature hashing** (Weinberger et al., 2009). Each feature is assigned an index using a hash function. If a hash function that permits collisions is chosen (typically by taking the hash output modulo some integer), then the model can be made arbitrarily small, as multiple features share a single weight. Because most features are rare, accuracy is surprisingly robust to such collisions (Ganchev and Dredze, 2008).

3.3.3 Count or binary?

Finally, we may consider whether we want our feature vector to include the **count** of each word, or its **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? Motivated by this intuition, Pang et al. (2002) use binary indicators of presence or absence in the feature vector: $f_j(\boldsymbol{x}, y) \in \{0, 1\}$. They find that classifiers trained on these binary vectors tend to outperform feature vectors based on word counts. One explanation is that words tend to appear in clumps: if a word has appeared once in a document, it is likely to appear again (Church, 2000). These subsequent appearances can be attributed to this tendency towards repetition, and thus provide little additional information about the class label of the document.

3.4 Evaluating classifiers

In any supervised machine learning application, 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 the estimated accuracy to be overly optimistic, because the classifier would be customized to this data, and would not perform as well as on unseen data in the future. It is usually necessary to set hyperparameters or perform feature selection, so you may need to construct a **tuning** or **development set** for this purpose. The development set should not intersect with the test data. For more details, see § 1.2.6.

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,

$$\text{acc}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \frac{1}{N} \sum_{i}^{N} \delta(y^{(i)} = \hat{y}). \tag{3.4}$$

If you've ever taken an exam, it was probably graded by accuracy. Why are other metrics necessary? The main reason is **class imbalance**. Suppose we were building a classifier to detect whether a electronic health record (EHR) described symptoms of a rare disease, which appear in only 1% of all documents in the dataset. A classifier that reports $\hat{y} = -1$ for all documents would achieve 99% accuracy, but would be practically useless. We need metrics that are capable of detecting the classifier's ability to discriminate between classes, even when the distribution is skewed.

One solution would be to build a **balanced test set**, in which 50% of documents are positive. But this would mean throwing away 98% of the original dataset! Furthermore, the detection threshold itself might be a design consideration: in health-related applications, we might prefer a very sensitive classifier, which returned a positive prediction if there is even a small chance that $y^{(i)} = +1$. In other applications, a positive result might trigger a costly action, so we would prefer a classifier that only makes positive predictions when absolutely certain. We need additional metrics to capture these characteristics.

3.4.1 Precision, recall, and F-measure

For any label (e.g., positive for presence of symptoms of a disease), there are two possible errors:

- **False positive**: the system incorrectly predicts the label.
- **False negative**: the system incorrectly fails to predict the label.

Similarly, for any label, there are two ways to be correct:

- **True positive**: the system correctly predicts the label.
- **True negative**: the system correctly predicts that the label does not apply to this instance.

Classifiers that make a lot of false positive errors are too sensitive; classifiers that make a lot of false negative errors are not sensitive enough. We can capture these two behaviors through two additional metrics, **recall** and **precision**:

$$\text{recall}(\boldsymbol{y}, \hat{\boldsymbol{y}}, k) = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{3.5}$$

$$\text{precision}(\boldsymbol{y}, \hat{\boldsymbol{y}}, k) = \frac{\text{TP}}{\text{TP} + \text{FP}}. \tag{3.6}$$

Recall and precision are both conditional likelihoods of a correct prediction, which is why their numerators are the same. Recall is conditioned on k being the correct label, $y^{(i)} = k$, so the denominator sums over true positive and false negatives. Precision is conditioned on k being the prediction, so the denominator sums over true positives and false positives. Note that true negatives are not considered in either statistic. The classifier that labels every document as "negative" would achieve zero recall; precision would be $\frac{0}{0}$.

Recall and precision are complementary objectives. A high-recall classifier is preferred when false negatives are cheaper than false positives: for example, in a preliminary screening for symptoms of a disease, the cost of a false positive might be an additional test, while a false negative would result in the disease going untreated. Conversely, we prefer a high-precision classifier when false positives are more expensive: for example, in spam detection, a false negative is a relatively minor inconvenience, while a false positive might mean that an important message goes unread.

The F**-measure** combines recall and precision into a single metric, using the harmonic mean:

$$F\text{-measure}(\boldsymbol{y}, \hat{\boldsymbol{y}}, k) = \frac{2rp}{r+p}, \tag{3.7}$$

where r is recall and p is precision.[6]

Evaluating multi-class classification Recall, precision, and F-measure are defined with respect to a specific label k. When there are multiple labels of interest (e.g., in word sense disambiguation), it is necessary to combine the F-measure across each class. **Macro F-measure** is the average F-measure across several classes,

$$\text{Macro-}F(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \frac{1}{|\mathcal{K}|}\sum_{k\in\mathcal{K}} F\text{-measure}(\boldsymbol{y}, \hat{\boldsymbol{y}}, k) \tag{3.8}$$

In multi-class problems with unbalanced class distributions, the macro F-measure is a balanced measure of how well the classifier recognizes each class. In **micro F-measure**, we compute true positives, false positives, and false negatives for each class, and then add them up to compute a single recall, precision, and F-measure. This metric is balanced across instances rather than classes, so it weights each class in proportion to its frequency — unlike macro F-measure, which weights each class equally.

3.4.2 Threshold-free metrics

In binary classification problems, it is possible to trade off between recall and precision by adding a constant "threshold" to the output of the scoring function. This makes it possible to trace out a curve, where each point indicates the performance at a single threshold. In the **receiver operating characteristic (ROC)** curve,[7], the x-axis indicates the **false positive rate**, $\frac{\text{FP}}{\text{FP+TN}}$, and the y-axis indicates the recall, or **true positive rate**. A perfect classifier attains perfect recall without any false positives, tracing a "curve" from the origin (0,0) to the upper left corner (0,1), and then to (1,1). In expectation, a non-discriminative classifier traces a diagonal line from the origin (0,0) to the upper right corner (1,1). Real classifiers tend to fall between these two extremes. Examples are shown in Figure 3.4.

[6] F-measure is sometimes called F_1, and generalizes to $F_\beta = \frac{(1+\beta^2)rp}{\beta^2 p + r}$. The β parameter can be tuned to emphasize recall or precision.

[7] The name "receiver operator characteristic" comes from the metric's origin in signal processing applications (Peterson et al., 1954). Other threshold-free metrics include precision-recall curves, precision-at-k, and balanced F-measure; see Manning et al. (2008) for more details.

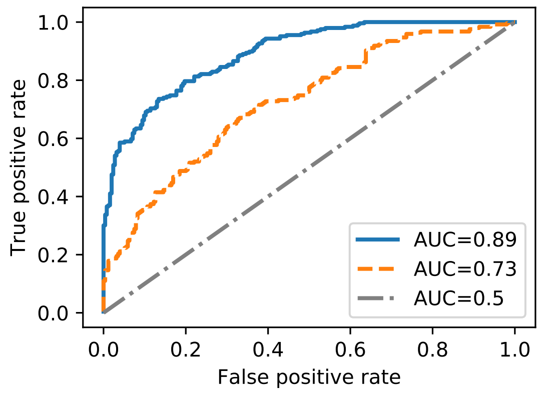


Figure 3.4: ROC curves for three classifiers of varying discriminative power, measured by AUC (area under the curve)

The ROC curve can be summarized in a single number by taking its integral, the **area under the curve (AUC)**. The AUC has an intuitive interpretation as the probability that a randomly-selected positive example will be assigned a higher score by the classifier than a randomly-selected negative example. Thus, a perfect classifier has AUC $= 1$ (all positive examples score higher than all negative examples); a random non-discriminative classifier has AUC $= 0.5$ (given a randomly selected positive and negative example, either could score higher with equal probability); a perfectly wrong classifier would have AUC $= 0$ (all negative examples score higher than all positive examples). One advantage of AUC in comparison to F-measure is that the baseline rate of 0.5 does not depend on the label distribution.

3.4.3 Classifier comparison and statistical significance

Building NLP systems often involves comparing different classification techniques. In some cases, the comparison is between algorithms, such as logistic regression versus averaged perceptron, or L_2 regularization versus L_1. In other cases, the comparison is between feature sets, such as the bag-of-words versus positional bag-of-words feature sets discussed in $\S$ 3.2.2. **Ablation testing** involves systematically removing (ablating) various aspects of the classifier, such as feature groups, and testing the **null hypothesis** that the ablated classifier is as good as the full model.

A full treatment of hypothesis testing is beyond the scope of this text, but this section contains a brief summary of the techniques necessary to compare classifiers. The main aim of hypothesis testing is to determine whether the difference between two statistics — for example, the accuracies of two classifiers — is likely to arise by chance. We will be concerned with chance fluctuations that arise due to the finite size of the test set.[8] An improvement of 10% on a test set with ten instances may reflect a random fluctuation that makes the test set more favorable to classifier c_1 than c_2; on another test set with a different ten instances, we might find that c_2 does better than c_1. But if we observe the same 10% improvement on a test set with 1000 instances, this is highly unlikely to be explained by chance. Such a finding is said to be **statistically significant** at a level p, which is the probability of an effect of equal or greater magnitude when the null hypothesis (that the classifiers are equally accurate) is true. For example, we write $p < .05$ when the likelihood of an equal or greater effect is less than 5%, assuming the null hypothesis is true.[9]

The binomial test

The statistical significance of a difference in accuracy can be evaluated using classical tests, such as the **binomial test**.[10] Suppose that classifiers c_1 and c_2 disagree on N instances in the test set, and that c_1 is correct on k of those instances. Under the null hypothesis that the classifiers are equally accurate, we would expect k/N to be roughly equal to $1/2$. As N increases, k/N should be increasingly close to $1/2$. These properties are captured by the **binomial distribution**, which is a probability over counts of binary random variables. We write $k \sim \text{Binom}(\theta, N)$ to indicate that k is drawn from a binomial distribution, with parameter N indicating the number of random "draws", and θ indicating the probability of "success" on each draw. The **probability mass function** (PMF) of the binomial distribution is,

$$\text{p}_{\text{Binom}}(k; N, \theta) = \binom{N}{k}\theta^k(1-\theta)^{N-k}, \tag{3.9}$$

with θ^k representing the probability of the k successes, $(1-\theta)^{N-k}$ representing the probability of the $N-k$ unsuccessful draws. The expression $\binom{N}{k} = \frac{N!}{k!(N-k)!}$ is a binomial coefficient, representing the number of possible orderings of events; this ensures that the distribution sums to one over all $k \in \{0, 1, 2, \ldots, N\}$.

Under the null hypothesis, $\theta = \frac{1}{2}$: when the classifiers disagree, they are each equally likely to be right. Now suppose that among N disagreements, c_1 is correct only $k <$

[8] Other sources of variance include the initialization of non-convex classifiers such as neural networks, and the ordering of instances in online learning such as stochastic gradient descent and perceptron.

[9] Statistical hypothesis testing is useful only to the extent that the existing test set is representative of the instances that will be encountered in the future. If, for example, the test set is constructed from news documents, no hypothesis test can predict which classifier will perform best on documents from another domain, such as electronic health records.

[10] A well-known alternative to the binomial test is **McNemar's test**, which computes a **test statistic** based on the number of examples that are correctly classified by one system and incorrectly classified by the other. The null hypothesis distribution for this test statistic is known to be drawn from a chi-squared distribution with a single degree of freedom, so a p-value can be computed from the cumulative density function of this distribution (Dietterich, 1998). Both tests give similar results in most circumstances, but the binomial test is easier to explain from first principles.

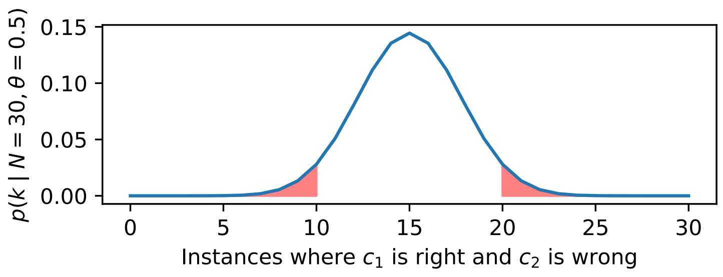


Figure 3.5: Probability mass function for the binomial distribution. The pink highlighted areas represent the cumulative probability for a significance test on an observation of $k = 10$ and $N = 30$.

$\frac{N}{2}$ times. The probability of c_1 being correct k or fewer times is the **one-tailed p-value**, because it is computed from the area under the binomial probability mass function from 0 to k, as shown in the left tail of Figure 3.5. This **cumulative probability** is computed as a sum over all values $i \leq k$,

$$\Pr_{\text{Binom}}\left(\text{count}(\hat{y}_2^{(i)} = y^{(i)} \neq \hat{y}_1^{(i)}) \leq k; N, \theta = \frac{1}{2}\right) = \sum_{i=0}^{k} p_{\text{Binom}}\left(i; N, \theta = \frac{1}{2}\right). \tag{3.10}$$

The one-tailed p-value applies only to the asymmetric null hypothesis that c_1 is at least as accurate as c_2. To test the **two-tailed** null hypothesis that c_1 and c_2 are equally accurate, we would take the sum of one-tailed p-values, where the second term is computed from the right tail of Figure 3.5. The binomial distribution is symmetric, so this can be computed by simply doubling the one-tailed p-value.

Two-tailed tests are more stringent, but they are necessary in cases in which there is no prior intuition about whether c_1 or c_2 is better. For example, in comparing logistic regression versus averaged perceptron, a two-tailed test is appropriate. In an ablation test, c_2 may contain a superset of the features available to c_1. If the additional features are thought to be likely to improve performance, then a one-tailed test would be appropriate, if chosen in advance. However, such a test can only prove that c_2 is more accurate than c_1, and not the reverse.

*Randomized testing

The binomial test is appropriate for accuracy, but not for more complex metrics such as F-measure. To compute statistical significance for arbitrary metrics, we must turn to randomization. Specifically, we draw a set of M **bootstrap samples** (Efron and Tibshirani, 1993), by resampling instances from the original test set with replacement. Each bootstrap sample is itself a test set of size N. Some instances from the original test set will

Algorithm 4 Bootstrap sampling for classifier evaluation. The original test set is $\{\boldsymbol{x}^{(1:N)}, \boldsymbol{y}^{(1:N)}\}$, the metric is $\delta(\cdot)$, and the number of samples is M.

```
procedure BOOTSTRAP-SAMPLE($\boldsymbol{x}^{(1:N)}, \boldsymbol{y}^{(1:N)}, \delta(\cdot), M$)
    for $t \in \{1, 2, \ldots, M\}$ do
        for $i \in \{1, 2, \ldots, N\}$ do
            $j \sim \text{UniformInteger}(1, N)$
            $\tilde{\boldsymbol{x}}^{(i)} \leftarrow \boldsymbol{x}^{(j)}$
            $\tilde{\boldsymbol{y}}^{(i)} \leftarrow \boldsymbol{y}^{(j)}$
        $d^{(t)} \leftarrow \delta(\tilde{\boldsymbol{x}}^{(1:N)}, \tilde{\boldsymbol{y}}^{(1:N)})$
    return $\{d^{(t)}\}_{t=1}^{M}$
```

not appear in any given bootstrap sample, while others will appear multiple times; but overall, the sample will be drawn from the same distribution as the original test set. We can then compute any desired evaluation on each bootstrap sample, which gives a distribution over the value of the metric. Algorithm 4 shows how to perform this computation.

To compare the F-measure of two classifiers c_1 and c_2, we set the function $\delta(\cdot)$ to compute the difference in F-measure on the bootstrap sample. If the difference is less than or equal to zero in at least 5% of the samples, then we cannot reject the one-tailed null hypothesis that c_2 is at least as good as c_1 (Berg-Kirkpatrick et al., 2012). We may also be interested in the 95% **confidence interval** around a metric of interest, such as the F-measure of a single classifier. This can be computed by sorting the output of Algorithm 4, and then setting the top and bottom of the 95% confidence interval to the values at the 2.5% and 97.5% percentiles of the sorted outputs. Alternatively, you can fit a normal distribution to the set of differences across bootstrap samples, and compute a Gaussian confidence interval from the mean and variance.

As the number of bootstrap samples goes to infinity, $M \rightarrow \infty$, the bootstrap estimate is increasingly accurate. A typical choice for M is 10^4 or 10^5; larger values are necessary for smaller p-values. One way to validate your choice of M is to run the test multiple times, and ensure that the p-values are similar; if not, increase M by an order of magnitude. This is a heuristic measure of the **variance** of the test, which can decreases with the square root $\sqrt{M}$ (Robert and Casella, 2013).

3.4.4 *Multiple comparisons

Sometimes it is necessary to perform multiple hypothesis tests, such as when comparing the performance of several classifiers on multiple datasets. Suppose you have five datasets, and you compare four versions of your classifier against a baseline system, for a total of 20 comparisons. Even if none of your classifiers is better than the baseline, there will be some chance variation in the results, and in expectation you will get one statistically significant improvement at $p = 0.05 = \frac{1}{20}$. It is therefore necessary to adjust the p-values when reporting the results of multiple comparisons.

One approach is to require a threshold of $\frac{\alpha}{m}$ to report a p value of $p < \alpha$ when performing m tests. This is known as the **Bonferroni correction**, and it limits the overall probability of incorrectly rejecting the null hypothesis at α. Another approach is to bound the **false discovery rate** (FDR), which is the fraction of null hypothesis rejections that are incorrect. Benjamini and Hochberg (1995) propose a p-value correction that bounds the fraction of false discoveries at α: sort the p-values of each individual test in ascending order, and set the significance threshold equal to largest k such that $p_k \leq \frac{k}{m}\alpha$. If $k > 1$, the FDR adjustment is more permissive than the Bonferroni correction.

3.5 Building datasets

For many classification tasks of interest, no labeled data exists. If you want to build a classifier, you must first build a dataset of your own. This includes selecting a set of documents or instances to annotate, and then performing the annotations.

In many cases, the scope of the dataset is determined by the application: if you want to build a system to classify electronic health records, then you must work with a corpus of records of the type that your classifier will encounter when deployed. In other cases, the goal is to build a system that will work across a broad range of documents. In this case, it is best to have a **balanced** corpus, with contributions from many styles and genres. For example, the Brown corpus draws from texts ranging from government documents to romance novels (Francis, 1964), and the Google Web Treebank includes annotations for five "domains" of web documents: question answers, emails, newsgroups, reviews, and blogs (Petrov and McDonald, 2012).

3.5.1 Metadata as labels

Annotation is difficult and time-consuming, and most people would rather avoid it. Luckily, it is sometimes possible to exploit existing metadata to obtain the desired labels. For example, reviews are often accompanied by a numerical rating, which can be converted into a classification label (see § 3.1). Similarly, the nationalities of social media users can be estimated from their profiles (Dredze et al., 2013) or even the time zones of their posts (Gouws et al., 2011). More ambitiously, we may try to classify the political affiliations of social media profiles based on their social network connections to politicians and major political parties (Rao et al., 2010).

The convenience of quickly constructing large labeled datasets without manual annotation is appealing. However this approach relies on the assumption that unlabeled instances — for which metadata is unavailable — will be similar to labeled instances. Consider the example of labeling the political affiliation of social media users based on their network ties to politicians. If a classifier attains high accuracy on such a test set, can we assume that it accurately predicts the political affiliation of all social media users? Probably not. Social media users who establish social network ties to politicians may also be more likely to mention politics in the text of their messages, as compared to the average user, for whom no political metadata is available. If so, the accuracy on a test set constructed from social network metadata would give an overly optimistic picture of the method's true performance on unlabeled data.

3.5.2 Labeling data

In many cases, there is no way to get ground truth labels other than manual annotation. Good annotations should satisfy several criteria: they should be **expressive** enough to capture the phenomenon of interest; they should be **replicable**, meaning that another annotator or team of annotators would produce very similar annotations if given the same data; and they should be **scalable**, so that they can be produced relatively quickly. Hovy and Lavid (2010) propose a structured procedure for obtaining annotations that meet these criteria, which is summarized below.

1. **Determine what the annotations are to include**. This is usually based on some theory of the underlying phenomenon: for example, if the goal is to produce annotations about the emotional state of a document's author, one should start with an theoretical account of the types or dimensions of emotion (e.g., Mohammad and Turney, 2013). At this stage, the tradeoff between expressiveness and scalability should be considered: a full instantiation of the underlying theory might be too costly to annotate at scale, so reasonable approximations should be considered.

2. Optionally, one may **design or select a software tool to support the annotation effort**. Existing general-purpose annotation tools include `BRAT` (Stenetorp et al., 2012) and `MMAX2` (Müller and Strube, 2006).

3. **Formalize the instructions for the annotation task**. To the extent that the instructions are not explicit, the resulting annotations will depend on the intuitions of the annotators. These intuitions may not be shared by other annotators, or by the users of the annotated data. Therefore explicit instructions are critical to ensuring the annotations are replicable and usable by other researchers.

4. **Perform a pilot annotation** of a small subset of data, with multiple annotators for each instance. This will give a preliminary assessment of both the replicability and scalability of the current annotation instructions. Metrics for computing the rate of agreement are described below. Manual analysis of specific disagreements should help to clarify the instructions, and may lead to modifications of the annotation task itself. For example, if two labels are commonly conflated by annotators, it may be best just to merge them.

5. After finalizing the annotation protocol and instructions, the main annotation effort can begin. Some if not all of the instances should receive multiple annotations, so that inter-annotator agreement can be computed. In some annotation projects, instances receive many annotations, which are then aggregated into a "consensus" label (e.g., Danescu-Niculescu-Mizil et al., 2013). However, if the annotations are time-consuming or require significant expertise, it may be preferable to maximize scalability by obtaining multiple annotations for only a small subset of examples.

6. Compute and report inter-annotator agreement, and release the data. In some cases, the raw text data cannot be released, usually due to concerns related to copyright or privacy. In these cases, one solution is to publicly release **stand-off annotations**, which contain links to document identifiers. The documents themselves can be released under the terms of a licensing agreement.

Measuring inter-annotator agreement

To measure the replicability of annotations, a standard practice is to compute the extent to which annotators agree with each other. If the annotators frequently disagree, this casts doubt on either their reliability or on the annotation system itself. For classification, one can compute the frequency with which the annotators agree; for rating scales, one can compute the average distance between ratings. These raw agreement statistics must then be compared with the rate of **chance agreement** — the level of agreement that would be obtained between two annotators who ignored the data.

Cohen's Kappa is widely used for quantifying the agreement on discrete labeling tasks (Cohen, 1960; Carletta, 1996),[11]

$$\kappa = \frac{\text{agreement} - E[\text{agreement}]}{1 - E[\text{agreement}]}. \tag{3.11}$$

The numerator is the difference between the observed agreement and the chance agreement, and the denominator is the difference between perfect agreement and chance agreement. Thus, $\kappa = 1$ when the annotators agree in every case, and $\kappa = 0$ when the annotators agree only as often as would happen by chance. Various heuristic scales have been proposed for determining when κ indicates "moderate", "good", or "substantial" agreement; for reference, Lee and Narayanan (2005) report $\kappa \approx 0.45 - 0.47$ for annotations of emotions in spoken dialogues, which they describe as "moderate agreement"; Stolcke et al. (2000) report $\kappa = 0.8$ for annotations of **dialogue acts**, which are labels for the purpose of each turn in a conversation.

When there are two annotators, the expected chance agreement is computed as,

$$E[\text{agreement}] = \sum_k \hat{\Pr}(Y = k)^2, \tag{3.12}$$

[11] For other types of annotations, Krippendorf's alpha is a popular choice (Hayes and Krippendorff, 2007; Artstein and Poesio, 2008).

where k is a sum over labels, and $\hat{\Pr}(Y = k)$ is the empirical probability of label k across all annotations. The formula is derived from the expected number of agreements if the annotations were randomly shuffled. Thus, in a binary labeling task, if one label is applied to 90% of instances, chance agreement is $.9^2 + .1^2 = .82$.

Crowdsourcing

Crowdsourcing is often used to rapidly obtain annotations for classification problems. For example, **Amazon Mechanical Turk** makes it possible to define "human intelligence tasks (hits)", such as labeling data. The researcher sets a price for each set of annotations and a list of minimal qualifications for annotators, such as their native language and their satisfaction rate on previous tasks. The use of relatively untrained "crowdworkers" contrasts with earlier annotation efforts, which relied on professional linguists (Marcus et al., 1993). However, crowdsourcing has been found to produce reliable annotations for many language-related tasks (Snow et al., 2008). Crowdsourcing is part of the broader field of **human computation** (Law and Ahn, 2011).

Exercises

1. As noted in § 3.3.3, words tend to appear in clumps, with subsequent occurrences of a word being more probable. More concretely, if word j has probability $\phi_{y,j}$ of appearing in a document with label y, then the probability of two appearances ($x_j^{(i)} = 2$) is greater than $\phi_{y,j}^2$.

   Suppose you are applying Naïve Bayes to a binary classification. Focus on a word j which is more probable under label $y = 1$, so that,

   $$\Pr(w = j \mid y = 1) > \Pr(w = j \mid y = 0). \tag{3.13}$$

   Now suppose that $x_j^{(i)} > 1$. All else equal, will the classifier overestimate or underestimate the posterior $\Pr(y = 1 \mid \boldsymbol{x})$?

2. Prove that F-measure is never greater than the arithmetic mean of recall and precision, $\frac{r+p}{2}$. Your solution should also show that F-measure is equal to $\frac{r+p}{2}$ iff $r = p$.

3. Given a binary classification problem in which the probability of the "positive" label is equal to α, what is the expected F-measure of a random classifier which ignores the data ($\hat{y} \perp y^{(i)}$)), and selects $\hat{y} = +1$ with probability $\frac{1}{2}$? (Assume that $\mathrm{p}(\hat{y}) \perp \mathrm{p}(y)$.) What is the expected F-measure of a classifier that selects $\hat{y} = +1$ with probability α (also independent of $y^{(i)}$)? Depending on α, which random classifier will score better?

4. Suppose that binary classifiers c_1 and c_2 disagree on $N = 30$ cases, and that c_1 is correct in $k = 10$ of those cases. Write a program that uses primitive functions such as $\exp$ and factorial to compute the **two-tailed** p-value — you may use an implementation of the "choose" function if one is avaiable. Verify your code against the output of a library for computing the binomial test or the binomial CDF, such as `scipy.stats.binom` in Python.

   Then use a randomized test to try to obtain the same p-value. In each sample, draw from a binomial distribution with $N = 30$ and $\theta = \frac{1}{2}$. Count the fraction of samples in which $k \leq 10$. This is the one-tailed p-value; double this to compute the two-tailed p-value.

   Try this with varying numbers of bootstrap samples: $M \in \{100, 1000, 5000, 10000\}$. For $M = 100$ and $M = 1000$, run the test 10 times, and plot the resulting p-values.

   Finally, perform the same tests for $N = 70$ and $k = 25$.

5. SemCor 3.0 is a labeled dataset for word sense disambiguation. You can download it,[12] or access it in `nltk.corpora.semcor`.

   Choose a word that appears at least ten times in SemCor (*find*), and annotate its WordNet senses across ten randomly-selected examples, without looking at the ground truth. Use online WordNet to understand the definition of each of the senses.[13] Have a partner do the same annotations, and compute the raw rate of agreement, expected chance rate of agreement, and Cohen's kappa.

6. Download the Pang and Lee movie review data, currently available from `http://www.cs.cornell.edu/people/pabo/movie-review-data/`. Hold out a randomly-selected 400 reviews as a test set.

   Download a sentiment lexicon, such as the one currently available from Bing Liu, `https://www.cs.uic.edu/~liub/FBS/sentiment-analysis.html`. Tokenize the data, and classify each document as positive iff it has more positive sentiment words than negative sentiment words. Compute the accuracy and F-measure on detecting positive reviews on the test set, using this lexicon-based classifier.

   Then train a discriminative classifier (averaged perceptron or logistic regression) on the training set, and compute its accuracy and F-measure on the test set.

   Determine whether the differences are statistically significant, using two-tailed hypothesis tests: Binomial for the difference in accuracy, and bootstrap for the difference in macro-F-measure.

[12] e.g., `https://github.com/google-research-datasets/word_sense_disambigation_corpora` or `http://globalwordnet.org/wordnet-annotated-corpora/`

[13] http://wordnetweb.princeton.edu/perl/webwn

The remaining problems will require you to build a classifier and test its properties. Pick a multi-class text classification dataset, such as RCV1[14]). Divide your data into training (60%), development (20%), and test sets (20%), if no such division already exists.

7. Compare various vocabulary sizes of $10^2, 10^3, 10^4, 10^5$, using the most frequent words in each case (you may use any reasonable tokenizer). Train logistic regression classifiers for each vocabulary size, and apply them to the development set. Plot the accuracy and Macro-F-measure with the increasing vocabulary size. For each vocabulary size, tune the regularizer to maximize accuracy on a subset of data that is held out from the training set.

8. Compare the following tokenization algorithms:

   - Whitespace, using a regular expression
   - Penn Treebank
   - Split input into five-character units, regardless of whitespace or punctuation

   Compute the token/type ratio for each tokenizer on the training data, and explain what you find. Train your classifier on each tokenized dataset, tuning the regularizer on a subset of data that is held out from the training data. Tokenize the development set, and report accuracy and Macro-F-measure.

9. Apply the Porter and Lancaster stemmers to the training set, using any reasonable tokenizer, and compute the token/type ratios. Train your classifier on the stemmed data, and compute the accuracy and Macro-F-measure on stemmed development data, again using a held-out portion of the training data to tune the regularizer.

10. Identify the best combination of vocabulary filtering, tokenization, and stemming from the previous three problems. Apply this preprocessing to the test set, and compute the test set accuracy and Macro-F-measure. Compare against a baseline system that applies no vocabulary filtering, whitespace tokenization, and no stemming.

    Use the binomial test to determine whether your best-performing system is significantly more accurate than the baseline.

    Use the bootstrap test with $M = 10^4$ to determine whether your best-performing system achieves significantly higher macro-F-measure.

[14]`http://www.ai.mit.edu/projects/jmlr/papers/volume5/lewis04a/lyrl2004_rcv1v2_README.htm`

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).

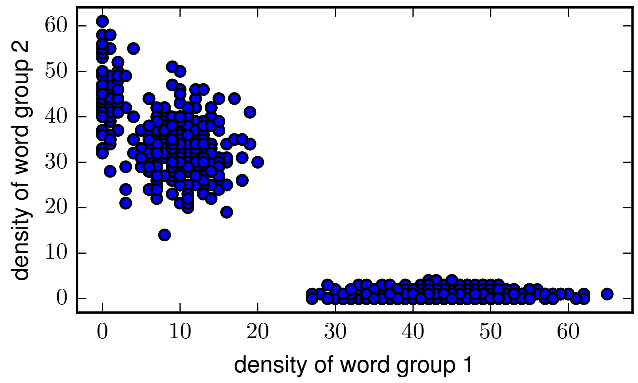


Figure 4.1: Counts of words from two different context groups

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 5.

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 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 $\boldsymbol{x}_i$

Algorithm 5 K-means clustering algorithm

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

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.

4.2.1 Step 1: the E-step

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

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

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

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.

4.2.2 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.

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).

4.2.3 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,

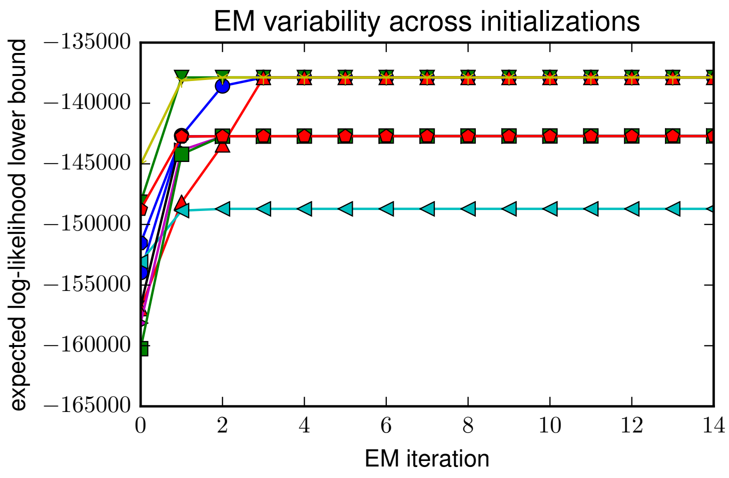


Figure 4.2: Sensitivity of expectation maximization to initialization

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 § 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.

4.2.4 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 (§ 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.

4.3.1 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.

4.3.2 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 **??**.

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:

$$\mathcal{L} = \log \mathrm{p}(\boldsymbol{x}^{(\ell)}, \boldsymbol{y}^{(\ell)}) + \lambda \log \mathrm{p}(\boldsymbol{x}^{(u)}) \tag{4.24}$$

$$\geq \log \mathrm{p}(\boldsymbol{x}^{(\ell)}, \boldsymbol{y}^{(\ell)}) + \lambda E_q[\log \mathrm{p}(\boldsymbol{x}^{(u)}, y)] \tag{4.25}$$

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.

4.3.3 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,

$$\log \mathrm{p}(\boldsymbol{x}_i, y_i) = \log \sum_{z}^{K_z} \mathrm{p}(\boldsymbol{x}_i, y_i, z) \tag{4.26}$$

$$\geq \log \mathrm{p}(y_i; \mu) + E_q \left[\log \mathrm{p}(\boldsymbol{x}_i \mid z; \boldsymbol{\phi}) + \log \mathrm{p}(z \mid y_i; \psi) - \log q_i(z) \right]. \tag{4.27}$$

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.

4.4.1 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,

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

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 $\boldsymbol{\phi}_y \mid \boldsymbol{x}, \boldsymbol{z} \sim \text{p}(\boldsymbol{\phi}_y \mid \boldsymbol{x}, \boldsymbol{z}, \alpha)$; 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).

4.4.2 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.

Part II

Sequences and trees

Chapter 5

Language models

In probabilistic classification, we are interested in computing the probability of a label, conditioned on the text. Let us now consider something like the inverse problem: computing the probability of text itself. Specifically, we will consider models that assign probability to a sequence of word tokens,[1] $\mathrm{p}(w_1, w_2, \ldots, w_M)$, with $w_m \in \mathcal{V}$. The set $\mathcal{V}$ is a discrete vocabulary,

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

Why would we want to compute the probaibility of a word sequence? In many applications, our goal is to produce word sequences as output:

- In **machine translation**, we convert from text in a source language to text in a target language.
- In **speech recognition**, we convert from audio signal to text.
- In **summarization**, we convert from long texts into short texts.
- In **dialogue systems**, we convert from the user's input (and perhaps an external knowledge base) into a text response.

In each of these cases, a key subcomponent is to compute the probability of the output text. By choosing high-probability output, we hope to generate texts that are more **fluent**. For example, suppose we want to translate a sentence from Spanish to English.

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

[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 standard written 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.

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

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

A good language model of English will tell us that the probability of this translation is low, in comparison with more grammatical alternatives, such as,

$$\text{p}(\textit{The coffee black me pleases much}) < \text{p}(\textit{I love dark coffee}). \tag{5.2}$$

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**, $\text{p}_e(\boldsymbol{w}^{(e)})$.
- The Spanish sentence $\boldsymbol{w}^{(s)}$ is then generated from a **translation model**, $\text{p}_{s|e}(\boldsymbol{w}^{(s)} \mid \boldsymbol{w}^{(e)})$.

Given these two distributions, we can then perform translation by Bayes rule:

$$\text{p}_{e|s}(\boldsymbol{w}^{(e)} \mid \boldsymbol{w}^{(s)}) \propto \text{p}_{e,s}(\boldsymbol{w}^{(e)}, \boldsymbol{w}^{(s)}) \tag{5.3}$$

$$= \text{p}_{s|e}(\boldsymbol{w}^{(s)} \mid \boldsymbol{w}^{(e)}) \times \text{p}_e(\boldsymbol{w}^{(e)}). \tag{5.4}$$

This is sometimes called the **noisy channel model**, because it envisions English text turning into Spanish by passing through a noisy channel, $\text{p}_{s|e}$. What is the advantage of modeling translation this way, as opposed to modeling $\text{p}_{e|s}$ directly? The crucial point is that the two distributions $\text{p}_{s|e}$ (the translation model) and p_e (the language model) can be estimated from separate data. The translation model requires **bitext** — examples of correct translations. But the language model requires only text in English. Such monolingual data is much more widely available, which means that the fluency of the output translation can be improved simply by scraping more webpages. Furthermore, once estimated, the language model p_e can be reused in any application that involves generating English text, from summarization to speech recognition.

5.1 N-gram language models

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:

$$\begin{aligned}&\text{p}(\textit{Computers are useless, they can only give you answers})\\&=\frac{\text{count}(\textit{Computers are useless, they can only give you answers})}{\text{count}(\text{all sentences ever spoken})}\end{aligned} \tag{5.5}$$

This estimator is **unbiased**: in the theoretical limit of infinite data, the estimate will be correct. 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 vocabularly for English would have $|\mathcal{V}| = 10^4$, so we would have 10^{80} possible sequences. Clearly, this estimator is very data-hungry, and suffers from high variance: even grammatical sentences will have probability zero if they happen not to have occurred in the training data.[2] We therefore 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.

We begin with n-gram language models, which compute the probability of a sequence as the product of probabilities of subsequences. The probability of a sequence $\text{p}(\boldsymbol{w}) = \text{p}(w_1, w_2, \ldots, w_M)$ can be refactored using the chain rule:

$$\text{p}(\boldsymbol{w}) = \text{p}(w_1, w_2, \ldots, w_M) \tag{5.6}$$

$$= \text{p}(w_1) \times \text{p}(w_2 \mid w_1) \times \text{p}(w_3 \mid w_2, w_1) \times \ldots \times \text{p}(w_M \mid w_{M-1}, \ldots, w_1) \tag{5.7}$$

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: given the context *Computers are*, we want to compute a probability over the next token. The relative frequency estimate of the probability of the word *useless* in this context is,

$$\begin{aligned}\text{p}(\textit{useless} \mid \textit{computers are}) &= \frac{\text{count}(\textit{computers are useless})}{\sum_{x \in \mathcal{V}} \text{count}(\textit{computers are } x)}\\&= \frac{\text{count}(\textit{computers are useless})}{\text{count}(\textit{computers are})}.\end{aligned}$$

Note that we haven't made any approximations yet, and we could have just as well applied the chain rule in reverse order,

$$\text{p}(\boldsymbol{w}) = \text{p}(w_M) \times \text{p}(w_{M-1} \mid w_M) \times \ldots \times \text{p}(w_1 \mid w_2, \ldots, w_M), \tag{5.8}$$

[2] Chomsky has famously argued that this is evidence against the very concept of probabilistic language models: no such model could distinguish the grammatical sentence *colorless green ideas sleep furiously* from the ungrammatical permutation *furiously sleep ideas green colorless*. Indeed, even the bigrams in these two examples are unlikely to occur — at least, not in texts written before Chomsky proposed this example.

or in any other order. But this means that we also haven't really improved anything either: to compute the conditional probability $\text{p}(w_M \mid w_{M-1}, w_{M-2}, \ldots, w_1)$, we need to model $|\mathcal{V}|^{M-1}$ contexts. We cannot estimate such a distribution from any reasonable finite sample.

To solve this problem, n-gram models make a crucial simplifying approximation: condition on only the past $n-1$ words.

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

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

$$\text{p}(w_1, \ldots, w_M) \approx \prod_m^M \text{p}(w_m \mid w_{m-1}, \ldots, w_{m-n+1}) \tag{5.10}$$

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:

$$\text{p}(\textit{I like black coffee}) = \text{p}(\textit{I} \mid \lozenge) \times \text{p}(\textit{like} \mid \textit{I}) \times \text{p}(\textit{black} \mid \textit{like}) \times \text{p}(\textit{coffee} \mid \textit{black}) \times \text{p}(\blacklozenge \mid \textit{coffee}). \tag{5.11}$$

In this model, we have to estimate and store the probability of only $|\mathcal{V}|^n$ events, which is exponential in the order of the n-gram, and not $|\mathcal{V}|^M$, which is exponential in the length of the sentence. The n-gram probabilities can be computed by relative frequency estimation,

$$\Pr(W_m = i \mid W_{m-1} = j, W_{m-2} = k) = \frac{\text{count}(i,j,k)}{\sum_{i'} \text{count}(i',j,k)} = \frac{\text{count}(i,j,k)}{\text{count}(j,k)} \tag{5.12}$$

A key design question is how to set the hyperparameter n, which controls the size of the context used in each conditional probability. If this is misspecified, the language model will sacrifice accuracy. Let's consider the potential problems concretely.

When n is too small. Consider the following sentences:

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

(5.4) *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 $|\mathcal{V}|^6$ events. Under a very small vocabulary of $|\mathcal{V}| = 10^4$, this means estimating the probability of 10^{24} distinct events.

These two problems point to another **bias-variance** tradeoff. A small n-gram size introduces high bias with respect to the true distribution, and a large n-gram size introduces high variance due to the huge number of possible events. But in reality the situation is even worse, because 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**.

5.2 Smoothing and discounting

Limited data is a persistent problem in estimating language models. In § 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).

5.2.1 Smoothing

A major concern in language modeling is to avoid the situation $\text{p}(\boldsymbol{w}) = 0$, 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 imaginary "pseudo" counts. The same idea can be applied to n-gram language models, as shown here in the bigram case,

$$\text{p}_{\text{smooth}}(w_m \mid w_{m-1}) = \frac{\text{count}(w_{m-1}, w_m) + \alpha}{\sum_{w' \in \mathcal{V}} \text{count}(w_{m-1}, w') + |\mathcal{V}|\alpha}. \tag{5.13}$$

This basic framework is called **Lidstone smoothing**, but special cases have other names:

- **Laplace smoothing** corresponds to the case $\alpha = 1$.
- **Jeffreys-Perks law** corresponds to the case $\alpha = 0.5$. Manning and Schütze (1999) offer more insight on the justifications for this setting.

			Lidstone smoothing, $\alpha = 0.1$		Discounting, $d = 0.1$	
	counts	unsmoothed probability	effective counts	smoothed probability	effective counts	smoothed probability
impropriety	8	0.4	7.826	0.391	7.9	0.395
offense	5	0.25	4.928	0.246	4.9	0.245
damage	4	0.2	3.961	0.198	3.9	0.195
deficiencies	2	0.1	2.029	0.101	1.9	0.095
outbreak	1	0.05	1.063	0.053	0.9	0.045
infirmity	0	0	0.097	0.005	0.25	0.013
cephalopods	0	0	0.097	0.005	0.25	0.013

Table 5.1: Example of Lidstone smoothing and absolute discounting in a bigram language model, for the context (*alleged*, _), for a toy corpus with a total of twenty counts over the seven words shown. Note that discounting decreases the probability for all but the unseen words, while Lidstone smoothing increases the effective counts and probabilities for *deficiencies* and *outbreak*.

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

$$c_i^* = (c_i + \alpha)\frac{M}{M + |\mathcal{V}|\alpha}, \tag{5.14}$$

where c_i is the count of event i, c_i^* is the effective count, and $M = \sum_i^{|\mathcal{V}|} c_i$ is the total number of terms in the dataset $(w_1, w_2, \ldots, w_M)$. This term ensures that $\sum_i^{|\mathcal{V}|} c_i^* = \sum_i^{|\mathcal{V}|} c_i = M$. The **discount** for each n-gram is then computed as,

$$d_i = \frac{c_i^*}{c_i} = \frac{(c_i + \alpha)}{c_i}\frac{M}{(M + |\mathcal{V}|\alpha)}.$$

5.2.2 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, suppose we set an absolute discount $d = 0.1$ in a bigram model, and then redistribute this probability mass equally over the unseen words. The resulting probabilities are shown in Table 5.1.

Discounting reserves some probability mass from the observed data, and we need not redistribute this 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, use unigrams. This is called **Katz backoff**. In this smoothing model, bigram probabilities are computed as,

$$c^*(i,j) = c(i,j) - d \tag{5.15}$$

$$\mathrm{p}_{\text{Katz}}(i \mid j) = \begin{cases} \frac{c^*(i,j)}{c(j)} & \text{if } c(i,j) > 0 \\ \alpha(j) \times \frac{\mathrm{p}_{\text{unigram}}(i)}{\sum_{i' : c(i',j)=0} \mathrm{p}_{\text{unigram}}(i')} & \text{if } c(i,j) = 0. \end{cases} \tag{5.16}$$

The term $\alpha(j)$ indicates the amount of probability mass that has been discounted for context j. This probability mass is then divided across all the unseen events, $\{i' : c(i',j) = 0\}$, proportional to the unigram probability of each word i'. The discount parameter d can be optimized to maximize performance (typically held-out log-likelihood) on a development set.

5.2.3 *Interpolation

Backoff is one way to combine n-gram models across various values of n. An alternative approach 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. For example, for an interpolated trigram model,

$$\begin{aligned} \mathrm{p}_{\text{Interpolation}}(i \mid j,k) = {} & \lambda_3 \mathrm{p}_3^*(i \mid j,k) \\ & + \lambda_2 \mathrm{p}_2^*(i \mid j) \\ & + \lambda_1 \mathrm{p}_1^*(i). \end{aligned}$$

In this equation, p_n^* is the unsmoothed empirical probability given by an n-gram language model, and λ_n is the weight assigned to this model. To ensure that the interpolated $\mathrm{p}(\boldsymbol{w})$ is still a valid probability distribution, we must obey the constraint, $\sum_n \lambda_n = 1$. But how to find the specific values of λ?

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. What's missing in this case? We can think of each word w_m as drawn from an n-gram of unknown size, $z_m \in \{1 \ldots n_{\text{max}}\}$. This z_m is the missing data that we are looking for. Therefore, the application of EM to this problem involves the following **generative process**:

- For each token $m \in \{1, 2, \ldots, M\}$:
  - draw $z_m \sim \text{Categorical}(\lambda)$,

– draw $w_m \sim \text{p}^*_{z_m}(w_m \mid w_{m-1}, \ldots w_{m-z_m})$.

If the missing data $\{Z_m\}$ were known, then we could estimate λ from relative frequency estimation,

$$\lambda_z = \frac{\text{count}(Z_m = z)}{M} \tag{5.17}$$

$$\propto \sum_{m=1}^{M} \delta(Z_m = z). \tag{5.18}$$

But since we do not know the values of the latent variables Z_m, we impute a distribution q_m in the E-step, which represents the degree of belief that word token w_m was generated from a n-gram of order z_m,

$$q_m(z) \triangleq \Pr(Z_m = z \mid \boldsymbol{w}_{1:m}; \lambda) \tag{5.19}$$

$$= \frac{\text{p}(w_m \mid \boldsymbol{w}_{1:m-1}, Z_m = z) \times \text{p}(z)}{\sum_{z'} \text{p}(w_m \mid \boldsymbol{w}_{1:m-1}, Z_m = z') \times \text{p}(z')} \tag{5.20}$$

$$\propto \text{p}^*_z(w_m \mid \boldsymbol{w}_{1:m-1}) \times \lambda_z. \tag{5.21}$$

In the M-step, we can compute λ by summing the expected counts under q,

$$\lambda_z \propto \sum_{m=1}^{M} q_m(z). \tag{5.22}$$

By iterating between updates to q and λ, we will ultimately converge at a solution. The complete algorithm is shown in Algorithm 6.

5.2.4 *Kneser-Ney smoothing

Kneser-Ney smoothing is based on absolute discounting, but it redistributes the resulting probability mass in a different way from Katz backoff. Empirical evidence points to Kneser-Ney smoothing as the state-of-art for n-gram language modeling **?**.

To motivate Kneser-Ney smoothing, consider the example: *I recently visited _*. Which of the following is more likely?

- *Francisco*
- *Duluth*

Now suppose that both bigrams *visited Duluth* and *visited Francisco* are unobserved in our training data, and furthermore, that the unigram probability p*(*Francisco*) is greater than p*(*Duluth*). Nonetheless we would still guess that p(*visited Duluth*) $>$ P(*visited Francisco*),

Algorithm 6 Expectation-maximization for interpolated language modeling

```
1: procedure ESTIMATE INTERPOLATED $n$-GRAM ($\boldsymbol{w}_{1:M}, \{\mathrm{p}^*_n\}_{n \in 1:n_{\max}}$)
2:     for $z \in \{1, 2, \ldots, n_{\max}\}$ do                          ▷ Initialization
3:         $\lambda_z \leftarrow \frac{1}{n_{\max}}$
4:     repeat
5:         for $m \in \{1, 2, \ldots, M\}$ do                              ▷ E-step
6:             for $z \in \{1, 2, \ldots, n_{\max}\}$ do
7:                 $q_m(z) \leftarrow \mathrm{p}^*_z(w_m \mid \boldsymbol{w}_{1:m-}) \times \lambda_z$
8:             $\boldsymbol{q}_m \leftarrow \text{Normalize}(\boldsymbol{q}_m)$
9:         for $z \in \{1, 2, \ldots, n_{\max}\}$ do                       ▷ M-step
10:            $\lambda_z \leftarrow \frac{1}{M} \sum_{m=1}^{M} q_m(z)$
11:    until tired
12:    return $\boldsymbol{\lambda}$
```

because *Duluth* is a more **versatile** word: it an occur in many contexts, while *Francisco* usually occurs in a single context, following the word *San*. This notion of versatility is the key to Kneser-Ney smoothing.

Writing u for a context of undefined length, and count(w, u) as the count of word w in context u, we define the Kneser-Ney bigram probability as

$$\mathrm{p}_{KN}(w \mid u) = \begin{cases} \frac{\text{count}(w,u)-d}{\text{count}(u)}, & \text{count}(w, u) > 0 \\ \alpha(u) \times \mathrm{p}_{\text{continuation}}(w), & \text{otherwise} \end{cases} \tag{5.23}$$

$$\mathrm{p}_{\text{continuation}}(w) = \frac{|u : \text{count}(w, u) > 0|}{\sum_{w' \in \mathcal{V}} |u' : \text{count}(w', u') > 0|}. \tag{5.24}$$

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 u is $d \times |w : \text{count}(w, u) > 0|$, and we divide this probability mass equally among the unseen n-grams,

$$\alpha(u) = |w : \text{count}(w, u) > 0| \times \frac{d}{\text{count}(u)}. \tag{5.25}$$

This is the amount of probability mass left to account for versatility, 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 u in which w appears, and in the denominator, we normalize by summing the same quantity over all words w'.

The idea of modeling versatility by counting contexts may seem heuristic, but there is an elegant theoretical justification from Bayesian nonparametrics (Teh, 2006). Kneser-Ney smoothing on n-grams was the dominant language modeling technique — widely used in speech recognition and machine translation — before the arrival of neural language models.

5.3 Recurrent neural network language models

Until this decade, n-grams were the dominant language modeling approach. But in a few years, they have been almost completely supplanted by a new family of language models based on **neural networks**. These models do not make the n-gram assumption of restricted context; indeed, they can incorporate arbitrarily distant contextual information, while remaining computationally and statistically tractable.

The first insight is to treat word prediction as a **discriminative** learning task: rather than directly estimating the distribution $\mathrm{p}(w \mid u)$ from (smoothed) relative frequencies, we now treat language modeling as a machine learning problem, and estimate parameters that maximize the log conditional probability of a corpus.[3]

The second insight is to reparametrize the probability distribution $\mathrm{p}(w \mid u)$ as a function of two dense K-dimensional numerical vectors, $\boldsymbol{\beta}_w \in \mathbb{R}^K$, and $\boldsymbol{v}_u \in \mathbb{R}^K$,

$$\mathrm{p}(w \mid u) = \frac{\exp(\boldsymbol{\beta}_w \cdot \boldsymbol{v}_u)}{\sum_{w' \in \mathcal{V}} \exp(\boldsymbol{\beta}_{w'} \cdot \boldsymbol{v}_u)}, \tag{5.26}$$

where $\boldsymbol{\beta}_w \cdot \boldsymbol{v}_u$ represents a dot product. Note that the denominator ensures that it is a properly normalized probability distribution. In the neural networks literature, this function is sometimes known as a **softmax** layer, written

$$(\mathrm{SoftMax}(\boldsymbol{a}))_i = \frac{\exp(a_i)}{\sum_j \exp(a_j)}, \tag{5.27}$$

where $\boldsymbol{a}$ is a vector of scores and $\mathrm{SoftMax}(\boldsymbol{a})_i$ is a normalized distribution.[4]

The word vectors $\boldsymbol{\beta}_w$ are parameters of the model, and are estimated directly. As we will see in chapter 14, these vectors carry useful information about word meaning, and semantically similar words tend to have highly correlated vectors.

The context vectors $\boldsymbol{v}_u$ can be computed in various ways, depending on the model. Here we will consider a relatively simple — but effective — neural language model, the **recurrent neural network** (RNN; Mikolov et al., 2010). The basic idea is to recurrently update the context vectors as we move through the sequence. Let us write $\boldsymbol{h}_m$ for the

[3]This idea is not in itself new; for example, Rosenfeld (1996) applies logistic regression to language modeling, and Roark et al. (2007) apply perceptrons and conditional random fields (§ 6.5.3).

[4]The logistic regression classifier can be viewed as an application of the softmax transformation to the vector constructed by computing the inner products of weights and features for all possible labels.

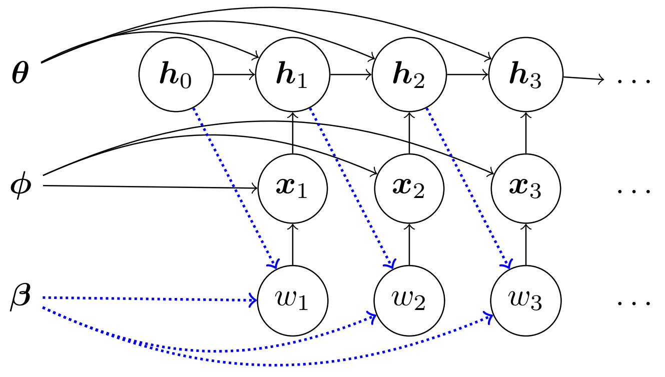


Figure 5.1: The recurrent neural network language model, viewed as an "unrolled" computation graph. Solid lines indicate direct computation, and dotted blue lines indicate probabilistic dependencies. Circles indicate variables; the left column of nodes are parameters.

contextual information at position m in the sequence. RNNs employ the following recurrence:

$$\boldsymbol{x}_m \triangleq \boldsymbol{\phi}_{w_m} \tag{5.28}$$

$$\boldsymbol{h}_m = g(\Theta \boldsymbol{h}_{m-1} + \boldsymbol{x}_m) \tag{5.29}$$

$$\mathrm{p}(w_{m+1} \mid w_1, w_2, \ldots, w_m) = \frac{\exp(\boldsymbol{\beta}_{w_{m+1}} \cdot \boldsymbol{h}_m)}{\sum_{w' \in \mathcal{V}} \exp(\boldsymbol{\beta}_{w'} \cdot \boldsymbol{h}_m)}, \tag{5.30}$$

where $\boldsymbol{\phi}$ is a matrix of **input word embeddings**, and $\boldsymbol{x}_m$ denotes the embedding for word w_m. The function g is an element-wise nonlinear **activation function**. Typical choices are:

- $\tanh(x)$, the hyperbolic tangent;
- $\sigma(x)$, the **sigmoid function** $\frac{1}{1+\exp(-x)}$;
- $(x)_+$, the **rectified linear unit**, $(x)_+ = \max(x, 0)$, also called **ReLU**.

These activation functions are shown in Figure 5.2. The sigmoid and tanh functions "squash" their inputs into a fixed range: $[0, 1]$ for the sigmoid, $[-1, 1]$ for $\tanh$. This makes it possible to chain together many instances of these functions without numerical instability.

A key point about the RNN language model is that although each w_m depends only on the context vector $\boldsymbol{h}_{m-1}$, this vector is in turn influenced by **all** previous tokens, $w_1, w_2, \ldots w_{m-1}$, through the recurrence operation: w_1 affects $\boldsymbol{h}_1$, which affects $\boldsymbol{h}_2$, and so on, until the information is propagated all the way to $\boldsymbol{h}_{m-1}$, and then on to w_m (see Figure 5.1). This is an important distinction from n-gram language models, where any information outside

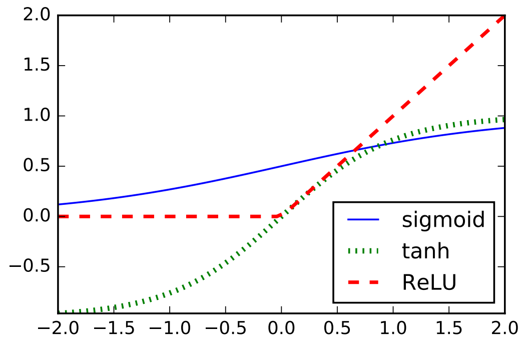


Figure 5.2: Nonlinear activation functions for neural networks

the n-word window is ignored. Thus, in principle, the RNN language model can handle long-range dependencies, such as number agreement over long spans of text — although it would be difficult to know where exactly in the vector $\boldsymbol{h}_m$ this information is represented. The main limitation is that information is attenuated by repeated application of the nonlinearity g. **Long short-term memories** (LSTMs), described below, are a variant of RNNs that address this issue, using memory cells to propagate information through the sequence without applying non-linearities (Hochreiter and Schmidhuber, 1997).

The denominator in Equation 5.30 is a computational bottleneck, because it involves a sum over the entire vocabulary. One solution is to use a **hierarchical softmax** function, which computes the sum more efficiently by organizing the vocabulary into a tree (Mikolov et al., 2011). Another strategy is to optimize an alternative metric, such as **noise-contrastive estimation** (Gutmann and Hyvärinen, 2012), which learns by distinguishing observed instances from artificial instances generated from a noise distribution (Mnih and Teh, 2012).

5.3.1 Estimation by backpropagation

The recurrent neural network language model has the following parameters:

- $\boldsymbol{\phi}_i \in \mathbb{R}^K$, the "input" word vectors (these are sometimes called **word embeddings**, since each word is embedded in a K-dimensional space);
- $\boldsymbol{\beta}_i \in \mathbb{R}^K$, the "output" word vectors;
- $\Theta \in \mathbb{R}^{K \times K}$, the recurrence operator.

Each of these parameters must be estimated. We do this by formulating an objective function over the training corpus, $\ell(\boldsymbol{w})$, and then employ **backpropagation** to incrementally update the parameters after encountering each training example. Backpropagation is a term from the neural network literature, which means that we use the chain rule of differentiation to obtain gradients on each parameter. After obtaining these gradients, we can apply an online learning algorithm such as stochastic gradient descent or adagrad, as discussed in § 2.4.2.

For example, suppose we want to obtain the gradient of the log-likelihood with respect to a single row of the recurrence operator, $\boldsymbol{\theta}_k$. Let us first define the total objective as a sum over local error functions e_m,

$$\ell(\boldsymbol{w}) = \sum_{m=1}^{M} e_m(\boldsymbol{h}_{m-1}) \tag{5.31}$$

$$e_m(\boldsymbol{h}_{m-1}) \triangleq -\log \mathrm{p}(w_m \mid w_1, w_2, \ldots, w_{m-1}) \tag{5.32}$$

$$= -\boldsymbol{\beta}_{w_m} \cdot \boldsymbol{h}_{m-1} + \log \sum_{w' \in \mathcal{V}} \exp\left(\boldsymbol{\beta}_{w'} \cdot \boldsymbol{h}_{m-1}\right). \tag{5.33}$$

We can now differentiate the objective with respect to $\boldsymbol{\theta}_k$:

$$\frac{\partial}{\partial \boldsymbol{\theta}_k} \ell(\boldsymbol{w}) = \sum_m \frac{\partial e_m(\boldsymbol{h}_{m-1})}{\partial \boldsymbol{\theta}_k} \tag{5.34}$$

$$= \sum_{m=1}^{M} (\nabla_{\boldsymbol{h}_{m-1}} e_m) \frac{\partial}{\partial \boldsymbol{\theta}_k} \boldsymbol{h}_{m-1}. \tag{5.35}$$

In the first line, we simply distribute the derivative across the sum. In the second line, we apply the chain rule of calculus. The term $\nabla_{\boldsymbol{h}_{m-1}} e_m$ refers to the gradient of the error e_m evaluated at $\boldsymbol{h}_{m-1}$, and is equal to,

$$\nabla_{\boldsymbol{h}_{m-1}} e_m = -\boldsymbol{\beta}_{w_m} + B\, \mathrm{SoftMax}(B \boldsymbol{h}_{m-1}), \tag{5.36}$$

where B is a matrix with all word output embeddings stacked vertically, $B = (\boldsymbol{\beta}_1^\top, \boldsymbol{\beta}_2^\top, \ldots, \boldsymbol{\beta}_{|\mathcal{V}|}^\top)$.

Next we compute the derivative of $\boldsymbol{h}_{m-1}$, first noting that within the vector $\boldsymbol{h}_{m-1}$, only the element $h_{m-1,k}$ depends on $\boldsymbol{\theta}_k$.

$$\frac{\partial}{\partial \boldsymbol{\theta}_k} \boldsymbol{h}_{m-1} = \frac{\partial}{\partial \boldsymbol{\theta}_k} h_{m-1,k} \tag{5.37}$$

$$= \frac{\partial}{\partial \boldsymbol{\theta}_k} g(\boldsymbol{\theta}_k \cdot \boldsymbol{h}_{m-2} + x_{m-1,k}) \tag{5.38}$$

$$= (\nabla_{\boldsymbol{\theta}_k \cdot \boldsymbol{h}_{m-2} + x_{m-1,k}} g) \times \frac{\partial}{\partial \boldsymbol{\theta}_k} (\boldsymbol{\theta}_k \cdot \boldsymbol{h}_{m-2} + x_{m-1,k}) \tag{5.39}$$

$$= (\nabla_{\boldsymbol{\theta}_k \cdot \boldsymbol{h}_{m-2} + x_{m-1,k}} g) \times (\boldsymbol{h}_{m-2} + \boldsymbol{\theta}_k \odot \frac{\partial}{\partial \boldsymbol{\theta}_k} \boldsymbol{h}_{m-2}), \tag{5.40}$$

where $\odot$ is an elementwise (Hadamard) vector product, and $(\nabla_{\boldsymbol{\theta}_k \cdot \boldsymbol{h}_{m-2} + x_{m-1,k}} g)$ is the elementwise gradient of the non-linear activation function for $\boldsymbol{h}_{m-1}$ evaluated at the scalar

$\boldsymbol{\theta}_k \cdot \boldsymbol{h}_{m-2} + x_{m-1,k}$. For example, if g is the elementwise hyperbolic tangent, then its gradient is,

$$(\nabla_{\boldsymbol{\theta}_k \cdot \boldsymbol{h}_{m-2} + x_{m-1,k}} g) = (1 - \tanh^2(\boldsymbol{\theta}_k \cdot \boldsymbol{h}_{m-2} + x_{m-1,k})). \tag{5.41}$$

The application of backpropagation to sequence models such as recurrent neural networks is known as **backpropagation through time**. A key point is that the derivative $\frac{\partial \boldsymbol{h}_{m-1}}{\partial \boldsymbol{\theta}_k}$ depends recurrently on $\frac{\partial \boldsymbol{h}_{m-2}}{\partial \boldsymbol{\theta}_k}$, and on all $\frac{\partial \boldsymbol{h}_n}{\partial \boldsymbol{\theta}_k}$ for $n < m$. Furthermore, we will need to compute $\frac{\partial \boldsymbol{h}_{m-2}}{\partial \boldsymbol{\theta}_k}$ **again**, to account for the error term $e_{m-1}(\boldsymbol{h}_{m-2})$. To avoid redoing work, it is best to cache such derivatives, so that they can be reused during backpropagation.

Backpropagation is implemented by neural network toolkits such as TensorFlow (Abadi et al., 2016), Torch (Collobert et al., 2011a), and DyNet (Neubig et al., 2017). In these toolkits, the user defines a **computation graph** representing the neural network structure, which culminates in a scalar loss function. The toolkit then automatically computes the gradient of the loss function with respect to all model parameters, by applying the chain rule of differentiation across the computation graph. Unlike the classification objectives considered in chapter 2, neural network objectives are usually non-convex function of the parameters, so there is no learning procedure that is guaranteed to converge to the global optimum. Nonetheless, gradient-based optimization often yields parameter estimates that are very effective in practice.

5.3.2 Hyperparameters

The RNN language model has several hyperparameters that must be tuned to ensure good performance. The model capacity is controlled by the size of the word and context vectors K, which play a role that is somewhat analogous to the size of the n-gram context. For datasets that are large with respect to the vocabulary (i.e., there is a large token-to-type ratio), we can afford to estimate a model with a large K, which enables more subtle distinctions between words and contexts. When the dataset is relatively small, then K must be smaller too. However, this general advice has not yet been formalized into any concrete formula for choosing K, and trial-and-error is still necessary. Overfitting can also be prevented by **dropout**, which involves randomly setting some elements of the computation to zero (Srivastava et al., 2014), forcing the learner not to rely too much on any particular dimension of the word or context vectors. (The dropout rate must also be tuned by the user.) Other design decisions include: the nature of the nonlinear activation function g, the size of the vocabulary, and the parametrization of the learning algorithm, such as the learning rate.

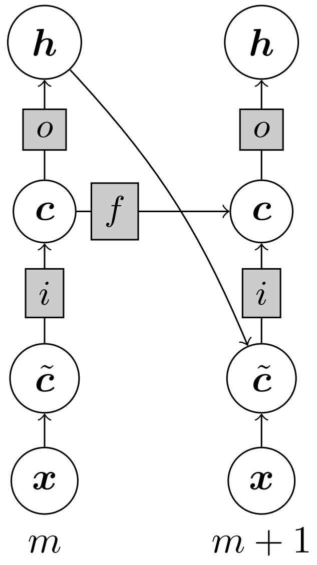


Figure 5.3: The long short-term memory (LSTM) architecture. For clarity, only the variables (and not the parameters) are shown. Gates are shown in shaded boxes. In an LSTM language model, each $\boldsymbol{h}_m$ would be used to predict the next word w_{m+1}.

5.3.3 Alternative neural language models

A well known problem with RNNs is that backpropagation across long chains tends to lead to "vanishing" or "exploding" gradients (Bengio et al., 1994). For example, the input embedding of word w_1 affects the likelihood of a distant word such as w_{29}, but this impact may be attenuated by backpropagation through the intervening time steps. One solution is to rescale the gradients, or to clip them at some maximum value (Pascanu et al., 2013). An alternative is to change the model architecture itself.

A popular variant of RNNs, which is more robust to these problems, is the **long short-term memory** (**LSTM**; Hochreiter and Schmidhuber, 1997; Sundermeyer et al., 2012). This model augments the hidden state $\boldsymbol{h}_m$ with a "memory cell" $\boldsymbol{c}_m$. The value of the memory cell at each time m is a linear interpolation between two quantities: its previous value $\boldsymbol{c}_{m-1}$, and an "update" $\tilde{c}_m$, which is computed from the current input $\boldsymbol{x}_m$ and the previous hidden state $\boldsymbol{h}_{m-1}$. The next state $\boldsymbol{h}_m$ is then computed from the memory cell. Because the memory cell is never passed through the non-linear function g, it is possible for information to propagate through the network over long distances.

The interpolation weights are controlled by a set of gates, which are themselves functions of the input and previous hidden state. The gates are computed from sigmoid activations, ensuring that their values will be in the range $[0, 1]$. They can therefore be viewed as soft, differentiable logic gates. The LSTM architecture is shown in Figure 5.3, and the complete update equations are:

$$\boldsymbol{f}_m = \sigma(\Theta^{(h\to f)} \cdot \boldsymbol{h}_{m-1} + \Theta^{(x\to f)} \cdot \boldsymbol{x}_m) \quad \text{forget gate} \tag{5.42}$$

$$\boldsymbol{i}_m = \sigma(\Theta^{(h\to i)} \cdot \boldsymbol{h}_{m-1} + \Theta^{(x\to i)} \cdot \boldsymbol{x}_m) \quad \text{input gate} \tag{5.43}$$

$$\tilde{\boldsymbol{c}}_m = \tanh(\Theta^{(h\to c)} \cdot \boldsymbol{h}_{m-1} + \Theta^{(w\to c)} \cdot \boldsymbol{x}_m) \quad \text{update candidate} \tag{5.44}$$

$$\boldsymbol{c}_m = \boldsymbol{f}_m \odot \boldsymbol{c}_{m-1} + \boldsymbol{i}_m \odot \tilde{\boldsymbol{c}}_m \quad \text{memory cell update} \tag{5.45}$$

$$\boldsymbol{o}_m = \sigma(\Theta^{(h\to o)} \cdot \boldsymbol{h}_{m-1} + \Theta^{(x\to o)} \cdot \boldsymbol{x}_m) \quad \text{output gate} \tag{5.46}$$

$$\boldsymbol{h}_m = \boldsymbol{o}_m \odot \boldsymbol{c}_m \quad \text{output.} \tag{5.47}$$

As above, $\odot$ refers to an elementwise (Hadamard) product. The LSTM model has been shown to outperform standard recurrent neural networks across a wide range of problems (it was first used for language modeling by Sundermeyer et al. (2012)), and is now widely used for sequence modeling tasks. There are several LSTM variants, of which the Gated Recurrent Unit (Cho et al., 2014b) is presently one of the more well known. Many software packages implement a variety of RNN architectures, so choosing between them is simple from a user's perspective. Jozefowicz et al. (2015) provide an empirical comparison of various modeling choices circa 2015. Notable earlier non-recurrent architectures include the neural probabilistic language model (Bengio et al., 2003) and the log-bilinear language model (Mnih and Hinton, 2007). Much more detail on these models can be found in the text by Goodfellow et al. (2016).

5.4 Evaluating language models

Because language models are typically components of larger systems — language modeling is not usually 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.

5.4.1 Held-out likelihood

The goal of probabilistic language models is to accurately measure the probability of sequences of word tokens. Therefore, an intrinsic evaluation metric is the likelihood that the language model assigns to **held-out data**, which is not used during training. Specifically, we compute,

$$\ell(\boldsymbol{w}) = \sum_{m=1}^{M} \log \mathrm{p}(w_m \mid w_{m-1}, \ldots), \tag{5.48}$$

treating the entire held-out corpus as a single stream of tokens.

Typically, unknown words are mapped to the $\langle \text{UNK} \rangle$ token. This means that we have to estimate some probability for $\langle \text{UNK} \rangle$ on the training data. One way to do this is to fix the vocabulary $\mathcal{V}$ to the $|\mathcal{V}| - 1$ words with the highest counts in the training data, and then convert all other tokens to $\langle \text{UNK} \rangle$. Other strategies for dealing with out-of-vocabulary terms are discussed in § 5.5.

5.4.2 Perplexity

Held-out likelihood is usually presented as **perplexity**, which is a deterministic transformation of the log-likelihood into an information-theoretic quantity,

$$\text{Perplex}(\boldsymbol{w}) = 2^{-\frac{\ell(\boldsymbol{w})}{M}}, \tag{5.49}$$

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

Lower perplexities correspond to higher likelihoods, so lower scores are better on this metric. (How to remember: lower perplexity is better, because your language model is less perplexed.) To understand perplexity, here are some special cases:

- In the limit of a perfect language model, probability 1 is assigned to the held-out corpus, with $\text{Perplex}(\boldsymbol{w}) = 2^{-\frac{1}{M}\log_2 1} = 2^0 = 1$.
- In the opposite limit, probability zero is assigned to the held-out corpus, which corresponds to an infinite perplexity, $\text{Perplex}(\boldsymbol{w}) = 2^{-\frac{1}{M}\log_2 0} = 2^{\infty} = \infty$.
- Assume a uniform, unigram model in which $\mathrm{p}(w_i) = \frac{1}{|\mathcal{V}|}$ for all words in the vocabulary. Then,

$$\begin{aligned}
\log_2(\boldsymbol{w}) =& \sum_{m=1}^{M} \log_2 \frac{1}{|\mathcal{V}|} = -\sum_{m=1}^{M} \log_2 |\mathcal{V}| = -M \log_2 |\mathcal{V}| \\
\text{Perplex}(\boldsymbol{w}) =& 2^{\frac{1}{M} M \log_2 |\mathcal{V}|} \\
=& 2^{\log_2 |\mathcal{V}|} \\
=& |\mathcal{V}|.
\end{aligned}$$

  This is the "worst reasonable case" scenario, since you could build such a language model without even looking at the data.

In practice, n-gram language models tend to give perplexities in the range between 1 and $|\mathcal{V}|$. For example, Jurafsky and Martin estimate a language model over a vocabularly of roughly $20,000$ words, on 38 million tokens of text from the Wall Street Journal (Jurafsky and Martin, 2009, page 97). They report the following perplexities on a held-out set of 1.5 million tokens:

- Unigram ($n = 1$): 962
- Bigram ($n = 2$): 170
- Trigram ($n = 3$): 109

Will this trend continue?

5.5 Out-of-vocabulary words

Through this chapter, we have assumed a **closed-vocabulary** setting — the vocabulary $\mathcal{V}$ is assumed to be a finite set. In realistic application scenarios, this assumption may not hold. Consider, for example, the problem of translating newspaper articles. The following sentence appeared in a Reuters article on January 6, 2017:[5]

> The report said U.S. intelligence agencies believe Russian military intelligence, the **GRU**, used intermediaries such as **WikiLeaks**, **DCLeaks.com** and the **Guccifer** 2.0 ”persona” to release emails...

Suppose that you trained a language model on the Gigaword corpus,[6] which was released in 2003. The bolded terms either did not exist at this date, or were not widely known; they are unlikely to be in the vocabulary. The same problem can occur for a variety of other terms: new technologies, previously unknown individuals, new words (e.g., *hashtag*), and numbers.

One solution is to simply mark all such terms with a special token, $\langle$UNK$\rangle$. While training the language model, we decide in advance on the vocabulary (often the K most common terms), and mark all other terms in the training data as $\langle$UNK$\rangle$. If we do not want to determine the vocabulary size in advance, an alternative approach is to simply mark the first occurrence of each word type as $\langle$UNK$\rangle$.

In some scenarios, we may prefer to make distinctions about the likelihood of various unknown words. This is particularly important in languages that have rich morphological systems, with many inflections for each word. For example, Spanish is only moderately complex from a morphological perspective, yet each verb has dozens of inflected forms.

[5]Bayoumy, Y. and Strobel, W. (2017, January 6). U.S. intel report: Putin directed cyber campaign to help Trump. *Reuters*. Retrieved from `http://www.reuters.com/article/us-usa-russia-cyber-idUSKBN14Q1T8` on January 7, 2017.

[6]`https://catalog.ldc.upenn.edu/LDC2003T05`

In such languages, there will necessarily be many word types that we do not encounter in a corpus, which are nonetheless predictable from the morphological rules of the language. To use a somewhat contrived English example, if *transfenestrate* is in the vocabulary, our language model should assign a non-zero probability to the past tense *transfenestrated*, even if it does not appear in the training data.

One way to accomplish this is to supplement word-level language models with **character-level language models**. Such models can use n-grams or RNNs, but with a fixed vocabulary equal to the set of ASCII or Unicode characters. For example Ling et al. (2015b) propose an LSTM model over characters, and Kim (2014) employ a **convolutional neural network** (LeCun and Bengio, 1995). A more linguistically motivated approach is to segment words into meaningful subword units, known as **morphemes** (see **??**). For example, Botha and Blunsom (2014) induce vector representations for morphemes, which they build into a log-bilinear language model; Bhatia et al. (2016) incorporate morpheme vectors into an LSTM.

Exercises

1. exercise tk

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 **N**OUNs, which describe things, properties, or ideas, and **V**ERBs, 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. We can write $\boldsymbol{f}((\boldsymbol{w}, m), y)$ to indicate the feature function for applying tag y to word w_m in the sequence $w_1, w_2, \ldots, w_M$. A simple tagging model would have a single base feature, the word itself:

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

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

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

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, with $\theta_{w,y} = 1$ for the tag y most frequently associated with 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 an animal rather than an 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:

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

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 **MODAL** 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 ADJECTIVE — 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

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

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 the label space $\mathcal{Y}(\boldsymbol{w}_{1:M})$,

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

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.

Given that the label space is exponentially large in the length of the sequence $w_1, \ldots, w_M$, can it ever be practical 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. 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 an amount of work that is exponential in the sequence length, so 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,

$$\boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}) = \sum_{m=1}^{M} \boldsymbol{f}(\boldsymbol{w}, y_m, m), \tag{6.8}$$

where we use the shorthand $\boldsymbol{w} \triangleq \boldsymbol{w}_{1:M}$. The summation in (6.8) means that the overall feature vector is the sum of feature vectors associated with each individual tagging decision. These 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.

$$\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}) = \boldsymbol{\theta} \cdot \sum_{m=1}^{M} \boldsymbol{f}(\boldsymbol{w}, y_m, m) \tag{6.9}$$

$$= \sum_{m=1}^{M} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, y_m, m) \tag{6.10}$$

$$\hat{y}_m = \operatorname*{argmax}_{y_m} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, y_m, m) \tag{6.11}$$

$$\hat{\boldsymbol{y}} = (\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_M) \tag{6.12}$$

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 $|\mathcal{Y}|^M$ 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,

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

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

$$\boldsymbol{f}(\boldsymbol{w} = \textit{they can fish}, \boldsymbol{y} = \text{N V V}) = \sum_{m=1}^{M} \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m) \tag{6.14}$$

$$\begin{aligned} =&\boldsymbol{f}(\boldsymbol{w}, \text{N}, \lozenge, 1) \\ &+ \boldsymbol{f}(\boldsymbol{w}, \text{V}, \text{N}, 2) \\ &+ \boldsymbol{f}(\boldsymbol{w}, \text{V}, \text{V}, 3) \end{aligned} \tag{6.15}$$

$$\begin{aligned} =&\langle w_m = \textit{they}, y_m = \text{N}\rangle + \langle y_m = \text{N}, y_{m-1} = \lozenge\rangle \\ &+ \langle w_m = \textit{can}, y_m = \text{V}\rangle + \langle y_m = \text{V}, y_{m-1} = \text{N}\rangle \\ &+ \langle w_m = \textit{fish}, y_m = \text{V}\rangle + \langle y_m = \text{V}, y_{m-1} = \text{V}\rangle \\ &+ \langle y_m = \blacklozenge, y_{m-1} = \text{V}\rangle. \end{aligned} \tag{6.16}$$

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.*

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,

$$\hat{\boldsymbol{y}} = \underset{\boldsymbol{y}}{\operatorname{argmax}}\, \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}) \tag{6.17}$$

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

Given this restriction on the feature function, we can solve this inference problem using **dynamic programming**, a algorithmic technique for reusing work in recurrent computations. As is often the case in dynamic programming, 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,

$$\max_{\boldsymbol{y}} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}) = \max_{\boldsymbol{y}_{1:M}} \sum_{m=1}^{M} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m) \tag{6.19}$$

$$= \max_{\boldsymbol{y}_{1:M}} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, y_M, y_{M-1}, M) + \sum_{m=1}^{M-1} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m) \tag{6.20}$$

$$= \max_{y_M} \max_{y_{M-1}} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, y_M, y_{M-1}, M) + \max_{\boldsymbol{y}_{1:M-2}} \sum_{m=1}^{M-1} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m). \tag{6.21}$$

In this derivation, we first removed the final element $\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, y_M, y_{M-1}, M)$ 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**,

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

Algorithm 7 The Viterbi algorithm.

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

where lower-case m indicates any position in the sequence, and $k \in \mathcal{Y}$ indicates a tag for that position. The variable $v_m(k)$ represents the score of the best tag sequence $(\hat{y}_1, \hat{y}_2, \dots, \hat{y}_m)$ that terminates in $\hat{y}_m = k$. 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,

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

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

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

Equation 6.26 is a **recurrence** for computing the Viterbi variables: each $v_m(k)$ 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 in a matrix, with the sequence index m on the columns, and the tag index k on the rows. In this representation, each $v_{m-1}(k)$ is connected to each $v_m(k')$, forming a **trellis**, as shown in Figure 6.1. As shown in the figure, special nodes are set aside for the start and end states.

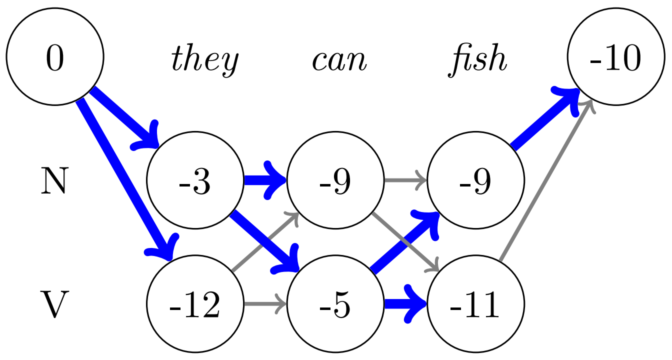


Figure 6.1: The trellis representation of the Viterbi variables, for the example *they can fish*, using the weights shown in Table 6.1.

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 problem 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 that tag k in position m. 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. These "back-pointers" are written $b_m(k)$, indicating the optimal tag y_{m-1} on the path to $Y_m = k$.

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 that we have $y_m = k$. Thus, we are looking for the tag $\hat{y}_{m-1}$ that maximizes,

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

$$= \underset{y_{m-1}}{\operatorname{argmax}}\, \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, k, y_{m-1}, m) + v_{m-1}(y_{m-1}), \tag{6.29}$$

which we obtain by plugging in the definition of the Viterbi variable. The value $\hat{y}_{m-1}$ was identified during forward pass, when computing the value of the Viterbi variable $v_m(k)$.

The complete Viterbi algorithm is shown in Algorithm 7. This formalizes the recurrences that were described in the previous 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 com-

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

(a) Weights for emission features.

	N	V	$\blacklozenge$
$\Diamond$	-1	-2	$-\infty$
N	-3	-1	-12
V	-1	-3	-1

(b) Weights for transition features. The "from" tags are on the columns, and the "to" tags are on the rows.

Table 6.1: Feature weights for the example trellis shown in Figure 6.1. Emission weights from $\Diamond$ and $\blacklozenge$ are implicitly set to $-\infty$.

puting 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, conjunctions are unlikely to end sentences in English, so we would like a large negative weight for the feature $\langle \text{CC}, \blacklozenge \rangle$; nouns are relatively likely to appear at the beginning of sentences, so we would like a more positive (or less negative) weight for the feature $\langle \Diamond, \text{N} \rangle$.

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. Computing each variable requires finding 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.

6.3.1 Example

To illustrate the Viterbi algorithm with an example, let us consider the minimal tagset $\{\text{N}, \text{V}\}$, corresponding to nouns and verbs. Even in this tagset, there is considerable ambiguity: for example, the words *can* and *fish* can each take both tags. Of the $2 \times 2 \times 2 = 8$ possible taggings for the sentence *they can fish*, four are possible given these possible tags, and two 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 7, 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, $(M + 1, \blacklozenge)$.

6.3.2 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 to generalize Viterbi to tag trigrams, which would involve the following feature decomposition:

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

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.

In the Cartesian product tag space, there are K^2 tags, suggesting that the time complexity will increase to $\mathcal{O}(MK^4)$. However, it is unnecessary to $\max$ over predecessor tag bigrams that are incompatible with the current tag bigram. By exploiting these constraints, it is possible to limit the time complexity to $\mathcal{O}(MK^3)$. The space complexity is $\mathcal{O}(MK^2)$. In general, the time and space complexity of higher-order Viterbi grows exponentially with the order of the tag n-grams that are considered in the feature decomposition.

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 $\text{p}(y \mid \boldsymbol{x}) \propto \text{p}(y, \boldsymbol{x})$. In probabilistic sequence labeling, our goal is similar: select the tag sequence that maximizes $\text{p}(\boldsymbol{y} \mid \boldsymbol{w}) \propto \text{p}(\boldsymbol{y}, \boldsymbol{w})$. Just as Naïve Bayes could be cast as a linear classifier maximizing $\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y)$, we can cast our probabilistic classifier as a linear decision rule. Furthermore, the feature restriction in (6.13) can be viewed as a conditional independence assumption on the random variables $\boldsymbol{y}$. Thanks to this assumption, it is possible to perform inference using the Viterbi algorithm.

Naïve Bayes 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, $\boldsymbol{y} \sim \text{p}(\boldsymbol{y})$; next, we draw the tokens from a conditional likelihood distribution, $\boldsymbol{w} \mid \boldsymbol{y} \sim \text{p}(\boldsymbol{w} \mid \boldsymbol{y})$.

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:

$$\mathrm{p}(\boldsymbol{w} \mid \boldsymbol{y}) = \prod_{m=1}^{M} \mathrm{p}(w_m \mid y_m). \tag{6.31}$$

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 $y_0 = \Diamond$ 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,

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

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 v_1 is probabilistically conditioned on another variable v_2, then there is an arrow $v_2 \rightarrow v_1$ in the diagram. If there are no arrows between v_1 and v_2, they are **conditionally independent**, given each variable's **Markov blanket**. In the hidden Markov model, the Markov blanket for each tag y_m includes the "parent" y_{m-1}, and the "children" y_{m+1} and w_m.[2]

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_n. However, if we are not given any information about the tags in a sequence, then all tags are probabilistically coupled.

6.4.1 Estimation

The hidden Markov model has two groups of parameters:

[2]In general graphical models, a variable's Markov blanket includes its parents, children, and its children's other parents (Murphy, 2012).

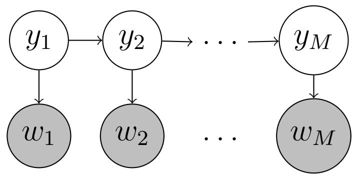


Figure 6.2: Graphical representation of the hidden Markov model. Arrows indicate probabilistic dependencies.

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,

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

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.

6.4.2 Inference

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

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

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 $\mathrm{p}(\boldsymbol{y}, \boldsymbol{w}) = \mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w}) \times \mathrm{p}(\boldsymbol{w}) \propto \mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w})$, which is proportional to the conditional probability. Therefore, we can reformulate the inference problem as,

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

We can now apply the HMM independence assumptions:

$$\log \mathrm{p}(\boldsymbol{y}, \boldsymbol{w}) = \log \mathrm{p}(\boldsymbol{y}) + \log \mathrm{p}(\boldsymbol{w} \mid \boldsymbol{y}) \tag{6.35}$$

$$= \sum_{m=1}^{M} \log \mathrm{p}_y(y_m \mid y_{m-1}) + \log \mathrm{p}_{w|y}(w_m \mid y_m) \tag{6.36}$$

$$= \sum_{m=1}^{M} \log \lambda_{y_m, y_{m-1}} + \log \phi_{y_m, w_m}. \tag{6.37}$$

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

$$\log \mathrm{p}(\boldsymbol{y}, \boldsymbol{w}) = \sum_{m=1}^{M} \log \lambda_{y_m, y_{m-1}} + \log \phi_{y_m, w_m} \tag{6.38}$$

$$= \sum_{m=1}^{M} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m), \tag{6.39}$$

where the feature function is defined,

$$\boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m) = \{\langle y_m, y_{m-1}\rangle, \langle y_m, w_m\rangle\}, \tag{6.40}$$

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 $\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m)$ can be interpreted probabilistically,

$$\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m) = \log \mathrm{p}_y(y_m \mid y_{m-1}) + \log \mathrm{p}_{w|y}(w_m \mid y_m) \tag{6.41}$$

$$= \log \mathrm{p}(y_m, w_m \mid y_{m-1}). \tag{6.42}$$

Now recall the definition of the Viterbi variables,

$$v_m(k) = \max_{\boldsymbol{y}_{1:m-1}} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, Y_m = k, y_{m-1}, m) + \sum_{n=1}^{m-1} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, y_n, y_{n-1}, n) \tag{6.43}$$

$$= \max_{\boldsymbol{y}_{1:m-1}} \log \mathrm{p}_{y_m, w_m | y_{m-1}}(k, w_m \mid y_{m-1}) + \sum_{n=1}^{m-1} \log \mathrm{p}(y_n, w_n \mid y_{n-1}) \tag{6.44}$$

$$= \max_{\boldsymbol{y}_{1:m-1}} \log \mathrm{p}(\boldsymbol{y}_{1:m-1}, Y_m = k, \boldsymbol{w}_{1:m}). \tag{6.45}$$

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

$$\max_{\boldsymbol{y}_{1:M}} \log \mathrm{p}(\boldsymbol{y}_{1:M}, \boldsymbol{w}_{1:M}) = \max_{y_M} \log \mathrm{p}_y(\blacklozenge \mid y_M) + v_M(y_M). \tag{6.46}$$

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

$$\begin{aligned}
v_m(k) =& \max_{\boldsymbol{y}_{1:m-1}} \mathrm{p}(\boldsymbol{y}_{1:m-1}, Y_m = k, \boldsymbol{w}_{1:m}) && (6.47)\\
=& \max_{y_{m-1}} \mathrm{p}(Y_m = k, w_m \mid y_{m-1}) \times \max_{\boldsymbol{y}_{1:m-2}} \mathrm{p}(y_{1:m-2}, y_{m-1}, \boldsymbol{w}_{1:m-1}) && (6.48)\\
=& \max_{y_{m-1}} \mathrm{p}(Y_m = k, w_m \mid y_{m-1}) \times v_{m-1}(y_{m-1}) && (6.49)\\
=& \mathrm{p}_E(w_m \mid Y_m = k) \times \max_{y_{m-1}} \mathrm{p}_T(y_m \mid y_{m-1}) \times v_{m-1}(y_{m-1}) && (6.50)\\
=& \max_{y_{m-1}} \exp(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, Y_m = k, y_{m-1}, m)) \times v_{m-1}(y_{m-1}). && (6.51)
\end{aligned}$$

In the final line, we use the fact that $\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m) = \log \mathrm{p}(y_m, w_m \mid y_{m-1})$, 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 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).

6.4.3 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 $\{\boldsymbol{w}^{(1)}, \boldsymbol{w}^{(2)}, \ldots, \boldsymbol{w}^{(N)}\}$, estimate the transition and emission distributions.

The Viterbi algorithm solves the decoding problem. We'll talk about the learning problem in § 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 § 6.5.3, but as a preview, we will use the following form,

$$\text{p}(y_m \mid \boldsymbol{w}_{1:M}) = \frac{\text{p}(y_m, \boldsymbol{w}_{1:M})}{\text{p}(\boldsymbol{w}_{1:M})}, \tag{6.52}$$

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**,

$$\alpha_m(k) \triangleq \text{p}(Y_m = k, \boldsymbol{w}_{1:m}). \tag{6.53}$$

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

$$\text{p}(\boldsymbol{w}) = \sum_{k \in \mathcal{Y}} \text{p}(Y_M = k, \boldsymbol{w}_{1:M}) \tag{6.54}$$

$$= \sum_{k \in \mathcal{Y}} \alpha_M(k). \tag{6.55}$$

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,

$$\alpha_m(k) = \text{p}(Y_m = k, \boldsymbol{w}_{1:m}) \tag{6.56}$$

$$= \text{p}(w_m \mid Y_m = k) \times \Pr(Y_m = k, \boldsymbol{w}_{1:m-1}) \tag{6.57}$$

$$= \text{p}(w_m \mid Y_m = k) \times \sum_{k' \in \mathcal{Y}} \Pr(Y_m = k, Y_{m-1} = k', \boldsymbol{w}_{1:m-1}) \tag{6.58}$$

$$= \text{p}(w_m \mid Y_m = k) \times \sum_{k' \in \mathcal{Y}} \Pr(Y_m = k \mid Y_{m-1} = k') \times \Pr(Y_{m-1} = k', \boldsymbol{w}_{1:m-1}) \tag{6.59}$$

$$= \text{p}(w_m \mid Y_m = k) \times \sum_{k' \in \mathcal{Y}} \Pr(Y_m = k \mid Y_{m-1} = k') \times \alpha_{m-1}(k') \tag{6.60}$$

$$= \sum_{k' \in \mathcal{Y}} \exp(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}_{1:M}, Y_m = k, Y_{m-1} = k', m)) \times \alpha_{m-1}(k'). \tag{6.61}$$

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 $\alpha_0(k) = \text{p}(Y_m = k \mid Y_{m-1} = \Diamond)$, to capture the transition probability from the start symbol. Comparing Equation 6.60 to Equation 6.50, 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,

$$\begin{aligned}
\alpha_m(k) \triangleq & \log \text{p}(\boldsymbol{w}_{1:m}, Y_m = k) && (6.62)\\
= & \log \sum_{k' \in \mathcal{Y}} \exp(\log \text{p}(w_m \mid Y_m = k) + \log \Pr(Y_m = k \mid Y_{m-1} = k') \\
& + \log \text{p}(\boldsymbol{w}_{1:m-1}, Y_{m-1} = k')) && (6.63)\\
= & \log \sum_{k' \in \mathcal{Y}} \exp(\log \text{p}(w_m \mid Y_m = k) + \log \Pr(Y_m = k \mid Y_{m-1} = k') + \alpha_{m-1}(k')) \\
& && (6.64)\\
= & \log \sum_{k' \in \mathcal{Y}} \exp(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}_{1:M}, Y_m = k, Y_{m-1} = k', m) + \alpha_{m-1}(k')). && (6.65)
\end{aligned}$$

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

6.4.4 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,

$$v_m(k) = \bigoplus_{k' \in \mathcal{Y}} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, Y_m = k, Y_{m-1} = k', m) \otimes v_{m-1}(k'). \qquad (6.66)$$

[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 $\overline{1}$, such that $a \oplus \overline{0} = a$ and $a \otimes \overline{1} = a$; and there must be a multiplicative annihilator $\overline{0}$, such that $a \otimes \overline{0} = \overline{0}$.

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.

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. 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 § 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 $\varnothing$ from *the*, indicating that this word has no suffix. The resulting feature vector is,

$$\begin{aligned}\boldsymbol{f}(\text{the slithy toves, D J N}) =& \{\langle W : \text{the}, \text{D}\rangle, \langle M : \varnothing, \text{D}\rangle, \langle T : \lozenge, \text{D}\rangle\\ & \langle W : \text{slithy}, \text{J}\rangle, \langle M : \text{-thy}, \text{J}\rangle, \langle T : \text{D}, \text{J}\rangle\\ & \langle W : \text{toves}, \text{N}\rangle, \langle M : \text{-es}, \text{N}\rangle, \langle T : \text{J}, \text{N}\rangle\\ & \langle T : \text{N}, \blacklozenge\rangle\}.\end{aligned}$$

[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.

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.

6.5.1 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:

$$\hat{y} = \underset{y \in \mathcal{Y}}{\operatorname{argmax}}\, \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y) \tag{6.67}$$

$$\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} + \boldsymbol{f}(\boldsymbol{x}, y) - \boldsymbol{f}(\boldsymbol{x}, \hat{y}). \tag{6.68}$$

We can apply exactly the same update in the case of structure prediction,

$$\hat{\boldsymbol{y}} = \underset{\boldsymbol{y} \in \mathcal{Y}(\boldsymbol{w})}{\operatorname{argmax}}\, \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}) \tag{6.69}$$

$$\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} + \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}) - \boldsymbol{f}(\boldsymbol{w}, \hat{\boldsymbol{y}}). \tag{6.70}$$

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 § 2.1.1).

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:

$$\theta_{\langle \textit{fish}, \mathrm{V} \rangle} \leftarrow \theta_{\langle \textit{fish}, \mathrm{V} \rangle} + 1 \tag{6.71}$$

$$\theta_{\langle \textit{fish}, \mathrm{N} \rangle} \leftarrow \theta_{\langle \textit{fish}, \mathrm{N} \rangle} - 1 \tag{6.72}$$

$$\theta_{\langle \mathrm{V}, \mathrm{V} \rangle} \leftarrow \theta_{\langle \mathrm{V}, \mathrm{V} \rangle} + 1 \tag{6.73}$$

$$\theta_{\langle \mathrm{V}, \mathrm{N} \rangle} \leftarrow \theta_{\langle \mathrm{V}, \mathrm{N} \rangle} - 1 \tag{6.74}$$

$$\theta_{\langle \mathrm{V}, \blacklozenge \rangle} \leftarrow \theta_{\langle \mathrm{V}, \blacklozenge \rangle} + 1 \tag{6.75}$$

$$\theta_{\langle \mathrm{N}, \blacklozenge \rangle} \leftarrow \theta_{\langle \mathrm{N}, \blacklozenge \rangle} - 1. \tag{6.76}$$

6.5.2 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,

$$\forall y \neq y^{(i)}, \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y^{(i)}) - \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y) \geq 1, \tag{6.77}$$

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 § 2.2.1).

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,

$$\forall \boldsymbol{y} \neq \boldsymbol{y}^{(i)}, \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)}) - \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}) \geq c(\boldsymbol{y}^{(i)}, \boldsymbol{y}). \tag{6.78}$$

This cost-augmented margin constraint specializes to the constraint in Equation 6.77 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**,

$$c(\boldsymbol{y}^{(i)}, \boldsymbol{y}) = \sum_{m=1}^{M} \delta(y_m^{(i)} \neq y_m). \tag{6.79}$$

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 $\hat{\boldsymbol{y}}$ that *maximally* violates

[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.

the margin constraint. We find this prediction by solving the following **cost-augmented decoding** problem:

$$\hat{\boldsymbol{y}} = \underset{\boldsymbol{y} \neq \boldsymbol{y}^{(i)}}{\operatorname{argmax}} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}) - \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)}) + c(\boldsymbol{y}^{(i)}, \boldsymbol{y}) \tag{6.80}$$

$$= \underset{\boldsymbol{y} \neq \boldsymbol{y}^{(i)}}{\operatorname{argmax}} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}) + c(\boldsymbol{y}^{(i)}, \boldsymbol{y}), \tag{6.81}$$

where in the second line we drop the term $\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)})$, which is constant in $\boldsymbol{y}$.

We can now formulate the margin constraint for sequence labeling,

$$\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)}) - \max_{\boldsymbol{y} \in \mathcal{Y}(\boldsymbol{w})} \Big(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}) + c(\boldsymbol{y}^{(i)}, \boldsymbol{y})\Big) \geq 0. \tag{6.82}$$

If the score for $\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)})$ 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, $\max_{\hat{\boldsymbol{y}}} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, \hat{\boldsymbol{y}})$, 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 c 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 large-margin classifiers, it is possible to formulate the learning problem in an unconstrained form, by combining a regularization term on the weights and a Lagrangian for the constraints:

$$\min_{\boldsymbol{\theta}} \quad \frac{1}{2}||\boldsymbol{\theta}||_2^2 - C\left(\sum_i \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)}) - \max_{\boldsymbol{y} \in \mathcal{Y}(\boldsymbol{w}^{(i)})} \Big[\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}) + c(\boldsymbol{y}^{(i)}, \boldsymbol{y})\Big]\right), \tag{6.83}$$

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

[6]To maximize over the set $\mathcal{Y}(\boldsymbol{w}) \setminus \boldsymbol{y}^{(i)}$ we would need 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.

proposed for structured support vector machines, some of which are discussed in § 2.2.1. 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.

6.5.3 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,

$$\mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w}) = \frac{\exp(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}))}{\sum_{\boldsymbol{y}' \in \mathcal{Y}(\boldsymbol{w})} \exp(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}'))}. \tag{6.84}$$

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

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 $\mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w})$,

$$\begin{aligned}
\hat{\boldsymbol{y}} &= \underset{\boldsymbol{y}}{\operatorname{argmax}} \, \mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w}) \\
&= \underset{\boldsymbol{y}}{\operatorname{argmax}} \log \mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w}) \\
&= \underset{\boldsymbol{y}}{\operatorname{argmax}} \, \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{y}, \boldsymbol{w}) - \log \sum_{\boldsymbol{y}' \in \mathcal{Y}(\boldsymbol{w})} e^{\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{y}', \boldsymbol{w})} \\
&= \underset{\boldsymbol{y}}{\operatorname{argmax}} \, \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{y}, \boldsymbol{w}).
\end{aligned}$$

This is identical to the decoding problem for structured perceptron, so the same Viterbi recurrence as defined in Equation 6.26 can be used.

[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.

Learning in CRFs

As with logistic regression, we learn the weights $\boldsymbol{\theta}$ by minimizing the regularized negative log conditional probability,

$$\ell = \frac{\lambda}{2}||\boldsymbol{\theta}||^2 - \sum_{i=1}^{N} \log \mathrm{p}(\boldsymbol{y}^{(i)} \mid \boldsymbol{w}^{(i)}; \boldsymbol{\theta}) \tag{6.85}$$

$$= \frac{\lambda}{2}||\boldsymbol{\theta}||^2 - \sum_{i=1}^{N} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)}) + \log \sum_{\boldsymbol{y}' \in \mathcal{Y}(\boldsymbol{w}^{(i)})} \exp\left(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}')\right), \tag{6.86}$$

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., 2011a), 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:

$$\frac{d\ell}{d\theta_j} = \lambda\theta_j + \sum_{i=1}^{N} E[f_j(\boldsymbol{w}^{(i)}, \boldsymbol{y})] - f_j(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)}), \tag{6.87}$$

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,

$$E[f_j(\boldsymbol{w}^{(i)}, \boldsymbol{y})] = \sum_{\boldsymbol{y} \in \mathcal{Y}(\boldsymbol{w}^{(i)})} \mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w}^{(i)}; \boldsymbol{\theta}) f_j(\boldsymbol{w}^{(i)}, \boldsymbol{y}) \tag{6.88}$$

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:

$$
\begin{aligned}
E[f_j(\boldsymbol{w}, \boldsymbol{y})] &= \sum_{\boldsymbol{y} \in \mathcal{Y}(\boldsymbol{w})} \mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w}; \boldsymbol{\theta}) f_j(\boldsymbol{w}, \boldsymbol{y}) && (6.89)\\
&= \sum_{\boldsymbol{y} \in \mathcal{Y}(\boldsymbol{w})} \mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w}; \boldsymbol{\theta}) \sum_{m=1}^{M} f_j(\boldsymbol{w}, y_m, y_{m-1}, m) && (6.90)\\
&= \sum_{m=1}^{M} \sum_{\boldsymbol{y} \in \mathcal{Y}(\boldsymbol{w})} \mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w}; \boldsymbol{\theta}) f_j(\boldsymbol{w}, y_m, y_{m-1}, m) && (6.91)\\
&= \sum_{m=1}^{M} \sum_{k,k'}^{\mathcal{Y}} \sum_{\boldsymbol{y} : y_{m-1}=k', y_m = k} \mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w}; \boldsymbol{\theta}) f_j(\boldsymbol{w}, k, k', m) && (6.92)\\
&= \sum_{m=1}^{M} \sum_{k,k'}^{\mathcal{Y}} f_j(\boldsymbol{w}, k, k', m) \sum_{\boldsymbol{y} : y_{m-1}=k', y_m = k} \mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w}; \boldsymbol{\theta}) && (6.93)\\
&= \sum_{m=1}^{M} \sum_{k,k'}^{\mathcal{Y}} f_j(\boldsymbol{w}, k', k, m) \Pr(Y_{m-1} = k', Y_m = k \mid \boldsymbol{w}; \boldsymbol{\theta}). && (6.94)
\end{aligned}
$$

This derivation works by interchanging the sum over tag sequences with the sum over indices m. At each position in the sequence, the locality restriction on features ensures that 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, the marginals can be computed efficiently using the **forward-backward algorithm**.

*Forward-backward algorithm

Recall that in the hidden Markov model, it was possible to use the forward algorithm to compute marginal probabilities $\mathrm{p}(y_m, \boldsymbol{w}_{1:m})$. We now derive a more general version of the forward algorithm, in which label sequences are scored in terms of **potentials** $\psi_m(k, k')$:

$$
\begin{aligned}
\psi_m(k, k') &\triangleq \exp(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, k, k', m)) && (6.95)\\
\mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w}) &= \frac{\prod_{m=1}^{M} \psi_m(y_m, y_{m-1})}{\sum_{\boldsymbol{y}'} \prod_{m=1}^{M} \psi_m(y'_m, y'_{m-1})}. && (6.96)
\end{aligned}
$$

Equation 6.96 simply expresses the CRF conditional likelihood, under a change of notation.

The tag bigram marginal probabilities can be written as,

$$\Pr(Y_{m-1} = k', Y_m = k \mid \boldsymbol{w}; \boldsymbol{\theta}) = \frac{\sum_{\boldsymbol{y}: y_m = k, y_{m-1} = k'} \prod_{n=1}^{M} \psi_n(y_n, y_{n-1})}{\sum_{\boldsymbol{y}'} \prod_{n=1}^{M} \psi_n(y'_n, y'_{n-1})}. \tag{6.97}$$

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 6.97, we sum over all tag sequences that include the transition $(Y_{m-1} = k') \to (Y_m = k)$. 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 $(Y_{m-1} = k') \to (Y_m = k)$, and the sum over **suffixes** $\boldsymbol{y}_{m:M}$,

$$\begin{aligned}\sum_{\boldsymbol{y}: Y_m = k, Y_{m-1} = k'} \prod_{n=1}^{M} \psi_n(y_n, y_{n-1}) = & \sum_{\boldsymbol{y}_{1:m-1}: y_{m-1} = k'} \prod_{n=1}^{m-1} \psi_n(y_n, y_n - 1) \\ & \times \psi_m(k, k') \\ & \times \sum_{\boldsymbol{y}_{m:M}: y_m = k} \prod_{n=m+1}^{M} \psi_n(y_n, y_{n-1}). \end{aligned} \tag{6.98}$$

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**,

$$\alpha_m(k) \triangleq \sum_{\boldsymbol{y}_{1:m}: y_m = k} \prod_{n=1}^{m} \psi_n(y_n, y_{n-1}) \tag{6.99}$$

$$= \sum_{k' \in \mathcal{Y}} \psi_m(k, k') \sum_{\boldsymbol{y}_{1:m-1}: y_{m-1} = k'} \prod_{n=1}^{m-1} \psi_n(y_n, y_{n-1}) \tag{6.100}$$

$$= \sum_{k' \in \mathcal{Y}} \psi_m(k, k') \times \alpha_{m-1}(k'). \tag{6.101}$$

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 § 6.4. If $\psi_m(k, k') = \mathrm{p}_E(w_m \mid Y_m = k) \times \mathrm{p}_T(k \mid k')$, then we exactly recover the Hidden Markov Model forward variable $\alpha_m(k) = \mathrm{p}(\boldsymbol{w}_{1:m}, Y_m = k)$ as computed in § 6.4.3.

[8] The terminology of "potentials" and "partition functions" comes from statistical mechanics (Bishop, 2006).

The third term of Equation 6.98 can also be defined recursively, this time moving over the trellis from right to left. The resulting recurrence is called the **backward algorithm**:

$$\beta_{m-1}(k) \triangleq \sum_{\boldsymbol{y}_{m-1:M}:y_{m-1}=k} \prod_{n=m}^{M} \psi_n(y_n, y_{n-1}) \tag{6.102}$$

$$= \sum_{k' \in \mathcal{Y}} \psi_m(k', k) \sum_{\boldsymbol{y}_{m:M}:y_m=k'} \prod_{n=m+1}^{M} \psi_n(y_n, y_{n-1}) \tag{6.103}$$

$$= \sum_{k' \in \mathcal{Y}} \psi_m(k', k) \times \beta_m(k'). \tag{6.104}$$

In practice, numerical stability requires that we use log-potentials rather than potentials, $\log \psi_m(y_m, y_{m-1}) = \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m)$. Then the sums must be replaced with log-sum-exp:

$$\log \alpha_m(k) = \log \sum_{k' \in \mathcal{Y}} \exp\left(\log \psi_m(k, k') + \log \alpha_{m-1}(k')\right) \tag{6.105}$$

$$\log \beta_{m-1}(k) = \log \sum_{k' \in \mathcal{Y}} \exp\left(\log \psi_m(k', k) + \log \beta_m(k')\right)). \tag{6.106}$$

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 (partition function), sometimes abbreviated as Z, can be written as,

$$Z \triangleq \sum_{\boldsymbol{y} \in \mathcal{Y}(\boldsymbol{w})} \mathrm{p}(\boldsymbol{w}, \boldsymbol{y}) \tag{6.107}$$

$$= \sum_{\boldsymbol{y} \in \mathcal{Y}(\boldsymbol{w})} \exp\left(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y})\right) \tag{6.108}$$

$$= \sum_{\boldsymbol{y} \in \mathcal{Y}(\boldsymbol{w})} \prod_{m=1}^{M} \exp\left(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m)\right) \tag{6.109}$$

$$= \sum_{\boldsymbol{y} \in \mathcal{Y}(\boldsymbol{w})} \prod_{m=1}^{M} \psi_m(y_m, y_{m-1}). \tag{6.110}$$

This term can be computed directly from either the forward or backward probabilities:

$$Z = \sum_{\boldsymbol{y} \in \mathcal{Y}(\boldsymbol{w})} \prod_{m=1}^{M} \psi_m(y_m, y_{m-1}) \tag{6.111}$$

$$= \alpha_{M+1}(\blacklozenge) \tag{6.112}$$

$$= \beta_0(\Diamond). \tag{6.113}$$

CRF learning: wrapup Having computed the forward and backward variables, we can compute the desired marginal probability as,

$$P(Y_{m-1} = k', Y_m = k \mid \boldsymbol{w}_{1:M}) = \frac{\alpha_{m-1}(k')\psi_m(k, k')\beta_m(k)}{Z}. \tag{6.114}$$

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]

6.5.4 Neural sequence labeling

Recently, neural network methods have been applied to sequence labeling. These methods can be employed in tandem with structure prediction algorithms such as Viterbi, although they are effective on their own. A relatively straightforward approach is to train a recurrent neural network or LSTM, as described in chapter 5; however, rather than predicting the next word in sequence, the model can be trained to predict the tag of the current word. A particularly effective approach is to train a **bidirectional recurrent neural network** (Graves and Schmidhuber, 2005), which estimates two hidden vectors, $\boldsymbol{h}_m = (\overrightarrow{\boldsymbol{h}}_m, \overleftarrow{\boldsymbol{h}}_m)$, with $\overrightarrow{\boldsymbol{h}}_m$ indicating the hidden state in a standard left-to-right RNN or LSTM, and $\overleftarrow{\boldsymbol{h}}_m$ indicating the hidden state in a left-to-right model. In this way, information from the entire sentence is brought to bear on the tagging decision for y_m. This approach was employed by Ling et al. (2015b), who find that bi-LSTMs perform considerably better than bi-RNNs on part-of-speech tagging, and that bidirectional variants of both models perform slightly but consistently better than their unidirectional counterparts. Note that LSTM-based tagging is a classification approach, so dynamic programming is not required to find the best tag sequence.

[9] The `CRFsuite` package implements several learning algorithms for CRFs (`http://www.chokkan.org/software/crfsuite/`).

It is also possible in neural sequence labeling to couple the tagging decisions, by introducing additional parameters for tag-to-tag transitions. Lample et al. (2016) dub this the **LSTM-CRF**, due to its combination of aspects of the long short-term memory and conditional random field models. They find that it improves performance on the task of **named entity recognition**, a sequence labeling task that is described in detail in the next chapter. This task has particularly strong dependencies between adjacent tags, so it is not surprising to see an advantage for structure predictions here.

Both Ling et al. (2015b) and Lample et al. (2016) find that it is advantageous to model unseen and rare words through their character-level representations. They train nested bi-LSTMs for each word, and take the concatenation of the introductory and final hidden states, $(\overrightarrow{\boldsymbol{h}}_M, \overleftarrow{\boldsymbol{h}}_0)$ as the word embeddings, which is then used as input in the tagging bi-LSTM. Lample et al. (2016) combine these character-level embeddings with **pre-trained** word embeddings, which were estimated from an unlabeled dataset that is many orders of magnitude larger than the labeled data. They do not backpropagate into the word embeddings, and learn only the transition and output parameters of the model.

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 6.101 and Equation 6.104. Because we are working in a hidden Markov model, we define the potentials as,

$$\psi_m(k, k') = \mathrm{p}_E(w_m \mid Y_m = k; \boldsymbol{\phi}) \times \mathrm{p}_T(Y_m = k \mid Y_{m-1} = k'; \lambda). \tag{6.115}$$

The expected counts are then,

$$E[\text{count}(W = i, Y = k)] = \sum_{m=1}^{M} \Pr(Y_m = k \mid \boldsymbol{w}_{1:M})\delta(W_m = i) \tag{6.116}$$

$$= \sum_{m=1}^{M} \frac{\Pr(Y_m = k, \boldsymbol{w}_{1:m})\text{p}(\boldsymbol{w}_{m+1:M} \mid Y_m = k)}{\text{p}(\boldsymbol{w}_{1:M})} \delta(W_m = i) \tag{6.117}$$

$$= \frac{1}{\alpha_{M+1}(\blacklozenge)} \sum_{m=1}^{M} \alpha_m(k)\beta_m(k)\delta(W_m = i) \tag{6.118}$$

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 $\text{p}(\boldsymbol{w}_{1:M}) = \alpha_{M+1}(\blacklozenge) = \beta_0(\lozenge)$.

The expected transition counts can be computed in a similar manner:

$$E[\text{count}(Y_m = k, Y_{m-1} = k')] = \sum_{m=1}^{M} \Pr(Y_m = k, Y_{m-1} = k' \mid \boldsymbol{w}_{1:M}) \tag{6.119}$$

$$\propto \sum_{m=1}^{M} \text{p}(Y_{m-1} = k', \boldsymbol{w}_{1:m-1})\text{p}(w_{m+1:M} \mid Y_m = k) \times \text{p}(w_m, Y_m = k \mid Y_{m-1} = k') \tag{6.120}$$

$$= \sum_{m=1}^{M} \text{p}(Y_{m-1} = k', \boldsymbol{w}_{1:m-1})\text{p}(w_{m+1:M} \mid Y_m = k) \times \text{p}(w_m \mid Y_m = k)\text{p}(Y_m = k \mid Y_{m-1} = k') \tag{6.121}$$

$$= \sum_{m=1}^{M} \alpha_{m-1}(k')\beta_m(k)\phi_{k,w_m}\lambda_{k' \to k}. \tag{6.122}$$

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 $\text{p}(\boldsymbol{w}_{1:M})$ to go from the joint probability $\text{p}(Y_{m-1} = k', Y_m = k, \boldsymbol{w}_{1:M})$ to the desired conditional $\Pr(Y_{m-1} = k', Y_m = k \mid \boldsymbol{w}_{1:M})$.

6.6.1 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 $\text{p}(y_m \mid \boldsymbol{x}_{1:M})$, 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).

6.6.2 Alternative unsupervised learning methods

As noted in § 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):

$$\text{p}(y_m \mid \boldsymbol{y}_{-m}, \boldsymbol{w}_{1:M}) \propto \text{p}(w_m \mid y_m)\text{p}(y_m \mid \boldsymbol{y}_{-m}). \tag{6.123}$$

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 $\text{p}(\boldsymbol{y}_{1:M} \mid \boldsymbol{w}_{1:M})$, 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).

Exercises

1. Consider the garden path sentence, *The old man the boat.* Given word-tag and tag-tag features, what inequality in the weights must hold for the correct tag sequence to outscore the garden path tag sequence for this example?

2. Sketch out an algorithm for a variant of Viterbi that returns the top-n label sequences. What is the time and space complexity of this algorithm?

3. Show how to compute the marginal probability $\Pr(Y_{m-2} = k, Y_m = k')$, in terms of the forwards and backward variables, and the potentials ψ_m.

4. more tk

Chapter 7

Applications of sequence labeling

Sequence labeling has applications throughout computational linguistics. This chapter focuses on the classical applications of part-of-speech tagging, shallow parsing, and named entity recognition, and touches briefly on two applications to interactive settings: dialogue act recognition and the detection of code-switching points between languages.

7.1 Part-of-speech tagging

The **syntax** of a language is the collection of principles under which sequences of words are judged to be grammatically acceptable by fluent speakers. One of the most basic syntactic concepts is the **part-of-speech** (POS), which refers to the syntactic role of each word in a sentence. We have already referred to this concept informally in the previous chapter, and you may have some intuitions from your own study of English. For example, in the sentence *Akanksha likes vegetarian sandwiches*, you may already know that *Akanksha* and *sandwiches* are nouns, *likes* is a verb, and *vegetarian* is an adjective.

Parts-of-speech can help to disentangle or explain various linguistic problems. Recall Chomsky's proposed distinction in chapter 5:

(7.1) Colorless green ideas sleep furiously.

(7.2) *Ideas colorless furiously green sleep.

Why is the first grammatical and the second not? One explanation is that the first example contains part-of-speech transitions that are typical in English: adjective to adjective, adjective to noun, noun to verb, and verb to adverb. In contrast, the second sentence contains transitions that are unusual: noun to adjective and adjective to verb. The ambiguity in sentences like *teacher strikes idle children* can also be explained in terms of parts of speech: in the interpretation that was likely intended, *strikes* is a noun and *idle* is a verb; in the alternative explanation, *strikes* is a verb and *idle* is an adjective.

Part-of-speech tagging is often taken as a early step in a natural language processing pipeline. Indeed, parts-of-speech provide features that can be useful for many of the tasks that we will encounter later, such as parsing, coreference resolution, and relation extraction. [todo: say more here]

7.1.1 Part-of-speech inventories

We have discussed a few parts-of-speech already: noun, verb, adjective, adverb. Jurafsky and Martin (2009) describe these as the four major **open word classes**, meaning that these are classes in which new words can be created.[1] These four open classes can be divided into more fine-grained subcategories, such as proper and common nouns; in addition, there are a number of closed classes. This section provides an overview of part-of-speech categories for English, but see Bender (2013) for a deeper linguistic perspective.

When creating a part-of-speech tagged dataset, the tagset inventory must be explicitly defined. The best known POS dataset for English is the Penn Treebank (PTB; Marcus et al., 1993), which includes a set of 45 tags. This chapter describes the linguistics of part-of-speech categories, and lists the corresponding PTB tags. The next section contrasts the PTB tagset with other tagsets for English and for other languages.

- **Nouns** describe entities and concepts.
  - **Proper nouns** name specific people and entities: e.g., *Georgia Tech*, *Janet*, *Buddhism*. In English, proper nouns are usually capitalized. The Penn Treebank (PTB) tags for proper nouns 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:

      (7.3) ***Fire*** *is dangerous.*

      (7.4) ***Skiing*** *is an expensive hobby.*

    In the Penn Treebank, singular and mass nouns are tagged NN, and plural nouns are tagged NNS.
  - **Pronouns** refer to specific entities or events that are already known to the reader or listener.
    - **Personal pronouns** refer to people or entities: *you*, *she*, *I*, *it*, *me*. The PTB tag is PRP.

[1]Languages need not have all four classes: for example, it has been argued that Korean does not have adjectives Kim (2002).

  * **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:

    (7.5) ***Where** are you going?*

    (7.6) *The man **who** wasn't there.*

    Possessive wh-pronouns (e.g., *The guy **whose** email got hacked*) are distinguished in the PTB with the tag WP$.

  Pronouns are considered a **closed class**, because unlike other nouns, it is generally not possible to introduce new pronouns.[2]

- **Verbs** describe activities, processes, and events. For example, *eat, write, sleep* are all verbs.

  – Just as nouns can be differentiated by number, verbs are differentiated by properties of the action they describe, and by their form. For English, the Penn Treebank differentiates the following types of verbs: VB (infinitive, e.g. *to shake*), VBD (past, e.g. *shook*), VBG (present participle, e.g. *shaking*), VBN (past participle, e.g. *shaken*), VBZ (present third-person singular, e.g. *shakes*), VBP (present, non-third-person singular, e.g. *shake*).

    Note that these verb classes include properties of the event like tense, as well as subject-verb agreement (third-person singular). Many languages have much more complex inflectional systems than English. For example, French verbs have unique inflections for every combination of person and number; when combined with features such as tense, mood, and aspect, there are several dozen possible inflections for each verb.

  – **Modals** are a closed subclass of verbs; they give additional information about the event described by the main verb of the sentence, e.g., *someone **should** do something*. In the PTB, their tag is MD. The PTB distinguishes modal verbs as all verbs which do not take a *-s* suffix in the third person singular present (Santorini, 1990).

  – The verb *to be* requires special treatment, as it must appear with a **predicative adjective** or noun, e.g.

    (7.7) *She is hungry.*

    (7.8) *We are Georgians.*

[2]Recent efforts to create gender-neutral singular pronouns in English are the exception that proves the rule: while battles over *s/he* and singular *they* have raged for decades, legions of new common nouns have been introduced (e.g., *selfie*, *emoji*) without much controversy.

The verbs *is* and *are* in these cases are called **copula**. The PTB does not distinguish copula from other verbs, but other tagsets do. More generally, in **light verb** constructions, the meaning is largely shaped by a predicative adjective or noun phrase, e.g. *he got fired*, *we took a walk*.

- **Auxiliary verbs** include *be, have, will*, which form complex tenses, negations, and questions.

(7.9) *They **had** spoken.*

(7.10) *She **did** not know.*

(7.11) ***Did** she know?*

(7.12) *We **will have** done it.*

Auxiliary verbs are not distinguished with special tags in the PTB; these cases are tagged as verbs, according to their person, number, and tense.

• **Adjectives** describe properties of entities. In English, adjectives can be used in two ways:

- **Attributive**: *an **antique** land*;
- **Predicative**: *the land was **antique***.

Adjectives (tagged JJ) may be **gradable**, meaning that they have a comparative form (e.g., *bigger*, *smellier*; tagged JJR) and superlative form (*biggest*, *smelliest*; tagged JJS). Adjectives like *antique* are not gradable.

• **Adverbs** describe properties of events. In the following examples, the bolded words are all adverbs.

(7.13) *He spoke **carefully**.*

(7.14) *She lives **downstairs**.*

(7.15) *I study **here**.*

(7.16) *Go **left** at the first traffic light.*

Adverbs are generally tagged RB, but **comparative adverbs** (e.g., *They played **harder***) are tagged RBR. Adverbs can describe a range of details about events:

- The **manner** of the event, e.g., *slowly, slower, fast, hesitantly*.
- The **degree** of the event, e.g., *extremely, very, highly*.
- Adverbs also include temporal information, such as *yesterday, Monday, soon*, and spatial information, such as *here*.

Adverbs do not only modify verbs; they may also modify sentences, adjectives, or other adverbs.

(7.17) ***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.

- **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,

  (7.18) *vast **and** trunkless legs*

  (7.19) *She plays backgammon **or** she does homework.*

  (7.20) *She eats **and** drinks quickly.*

  (7.21) *Sandeep lives north of Midtown **and** south of Buckhead.*

  (7.22) *Max cooked, **and** Abigail ate, all the pizza.*

- **Subordinating conjunctions** introduce a subordinate clause, e.g.

  (7.23) *She thinks **that** Chomsky is wrong about language models.*

  In the PTB, subordinating conjunctions are grouped with prepositions in the tag IN.

- **Particles** are words that travel combine with verbs to create **phrasal verbs** with meaning that is distinct from the verb alone, e.g.,

  (7.24) *Come **on**.*

  (7.25) *He brushed himself **off**.*

  (7.26) *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*. These words describe the **information status** and **uniqueness** of the noun phrase that follows. For example, *the book* refers to a unique entity, which is already known to the listener; *a book* can refer to the existence of an entity (*Tahira loaned him a book.*)

  - 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?*
  - **Pre-determiners** appear in constructions like ***both** my children*, where *my* is the central determiner.

  Of these determiners, only wh-determiners and pre-determiners are distinguished with special tags, WDT and PDT. All others are tagged DT.

- Finally, there are a few oddball categories.
  - **Existential there**, e.g. ***There** is no way out of here*, gets its own tag, EX.
  - The possessive ending *'s* is annotated POS. This means that we assume we have separated this ending into a separate token.
  - Other special tags are reserved for numbers (CD), foreign words (FW), list items (LS), interjections (e.g., *uh, wow*, tagged UH), and a range of non-alphabetic symbols (commas, dollar signs, quotation marks, etc.)

7.1.2 Tagset granularity

The previous section illustrates some of the design decisions that were made in creating the Penn Treebank tagset. In general, there is a tradeoff between capturing important linguistic details, and choosing a tagset that can annotated efficiently. The PTB strikes one balance; other tagsets make other decisions. In English, the Brown corpus favored a more granular tagset, with 87 part-of-speech tags (Francis, 1964)[3]. It has:

- 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*);
- distinct tags for the third-person singular pronouns (e.g., *it, he*) and other pronouns (e.g., *they, we, I*).

In the other direction, Petrov et al. (2012) propose a "universal" set of twelve part-of-speech tags, which apply across many languages (Petrov et al., 2012). 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;

[3]See `http://www.comp.leeds.ac.uk/ccalas/tagsets/brown.html` for the tagset details.

- preposition and postpositions are grouped as "adpositions";
- all punctuation is grouped;
- coordinating and subordinating conjunctions (e.g. *and* versus *that*) are grouped.

In the Universal Dependency Treebank (Nivre et al., 2016), the coarse-grained universal part-of-speech tags are augmented with **lexical features**, such as gender, number, case, and tense. These features are annotated only for languages in which they apply: for example, gender would be annotated for determiners in Spanish, but not in English.

How to decide among these tagging strategies? Each has its own advantages. 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.

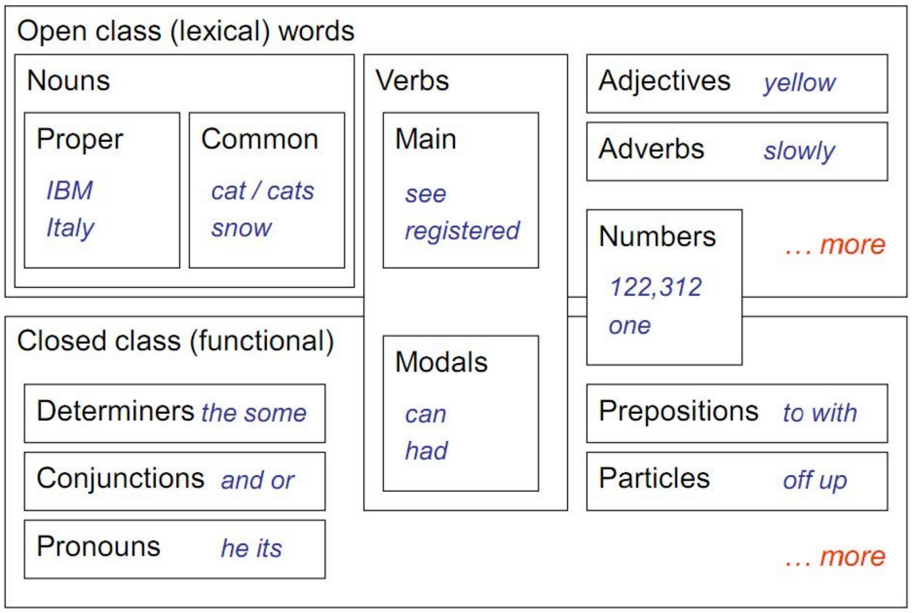


Figure 7.1: [todo: attribution?]

Example

To understand the linguistic differences between these tagsets, let's look at an example:

(7.27) 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.

	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.

7.1.3 Accurate part-of-speech tagging

Part-of-speech tagging is the problem of selecting the correct tag for each word in a sentence. Success is typically measured by accuracy on an annotated test set, which is simply the fraction of tokens that were tagged correctly. POS tagging has been a benchmark problem in natural language processing since the mid-1990s.

Baselines

A simple baseline for part-of-speech tagging is to choose the most common tag for each word. For example, in the Universal Dependency treebank, the word *talk* appears 96 times, and 85 of those times it is labeled as a verb: therefore, we will always predict verb when we see it in the test set. For words that do not appear in the training corpus, the baseline simply guesses the most common tag overall, which is noun. In the Penn Treebank, this simple baseline obtains accuracy above 92%. So the interesting question is how many of the remaining 8% of instances can be classified correctly.

Hidden Markov models 6.4 were a popular choice for part-of-speech tagging: the emission distribution links tags to words, and the transition distribution captures tag ordering constraints. However, hidden Markov models can struggle with rare words, which may not appear in the training data. For these words, it is important to use word-internal information, such as suffixes and capitalization; despite some heroic efforts (Brants, 2000), these features are difficult to incorporate into hidden Markov models.

Structure prediction

Conditional random fields and structured perceptron perform at or near the state-of-the-art for part-of-speech tagging in English. For example, (Collins, 2002) achieved 97.1% accuracy on the Penn Treebank, using a structured perceptron, using the following base features (originally introduced by Ratnaparkhi (1996)):

- current word, w_m
- previous words, w_{m-1}, w_{m-2}
- next words, w_{m+1}, w_{m+2}
- previous tag, y_{m-1}
- previous two tags, (y_{m-1}, y_{m-2})
- for rare words:
  - first k characters, up to $k = 4$
  - last k characters, up to $k = 4$
  - whether w_m contains a number, uppercase character, or hyphen.

A group from Stanford used a similar set of features in a model that is closely related to conditional random fields (Toutanova et al., 2003), attaining 97.3% accuracy. Eight years later, Chris Manning (one of the authors on the Stanford paper), remarked on how difficult it had been to obtain further improvements from either better features or machine learning models (Manning, 2011). Manning notes that despite the seemingly impressive 97% accuracy on the token level, only a little more than half of all sentences are tagged

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

completely correctly, suggesting that the task is far from solved. He suggests that better annotations may be necessary to drive further improvements.

Neural part-of-speech tagging

[todo: remind readers about LSTMs from § 5.3. Or maybe put the modeling stuff in the previous chapter?]

- Simple LSTM tagging, using pre-trained word embeddings Huang et al. (2015); Ma and Hovy (2016)
- Bi-directional LSTM (Graves et al., 2013)
- LSTM-CRF (Huang et al., 2015; Ma and Hovy, 2016). Improvements on PTB are small (error rate from 2.7% to 2.4%). Bigger improvements for NER

7.2 Tokenization

not sure whether to include this, but a major cite seems to be Peng et al. (2004).

7.3 Shallow parsing

7.4 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 17.1. 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 17.1. This is called BILOU notation, and has been shown to yield improvements in supervised named entity recognition (Ratinov and Roth, 2009).

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 17.1 include:

$$\begin{aligned}
\langle &\text{CURR-WORD:}\textit{Army},\\
&\text{PREV-WORD:}\textit{U.S.},\\
&\text{NEXT-WORD:}\textit{captured},\\
&\text{PREFIX-1:}\textit{A-},\\
&\text{PREFIX-2:}\textit{Ar-},\\
&\text{SUFFIX-1:}\textit{-y},\\
&\text{SUFFIX-2:}\textit{-my},\\
&\text{SHAPE:}\textit{Xxxx}\rangle
\end{aligned}$$

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).

7.5 Dialogue acts

[todo: define problem] (Stolcke et al., 2000; Galley, 2006)

7.6 Code switching

[todo: define problem] Solorio and Liu (2008) use a classification-based approach. Nguyen and Dogruöz23 (2013) use a conditional random field.

Chapter 8

Finite-state automata

Consider the following problems:

- Segment a word into its stem and affixes: *impossibility* $\rightarrow$ *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. $\{a,b,c,\ldots,z\}$, or $\{aardvark, abacus, \ldots, zyxt\}$.
- 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?

8.1.1 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:

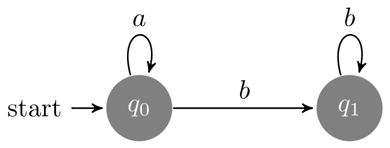
- a finite alphabet Σ of input symbols;
- a finite set of **states** $Q = \{q_0, q_1, \ldots, q_n\}$;
- a **start state** $q_0 \in Q$;
- a set of **final states** $F \subseteq Q$;
- a **transition function** $\delta : Q \times \Sigma \to 2^Q$. The transition function maps from a state and an input symbol 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.

[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.

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 $\delta : Q \times \Sigma \to Q$. This means that every pair of initial state and symbol can transition to at most one resulting state.
- In a nondeterministic (N)FSA, $\delta : Q \times \Sigma \to 2^Q$. 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).

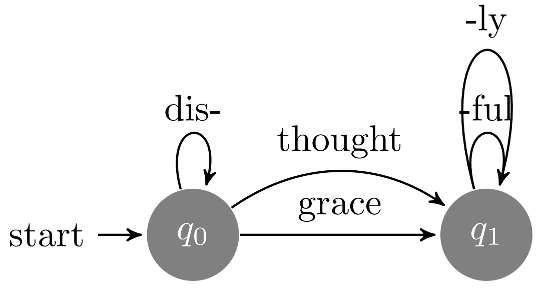


Figure 8.1: First try at modeling English morphology

8.1.2 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 $F = \{q_1\}$, 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 **grace-lyful* 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

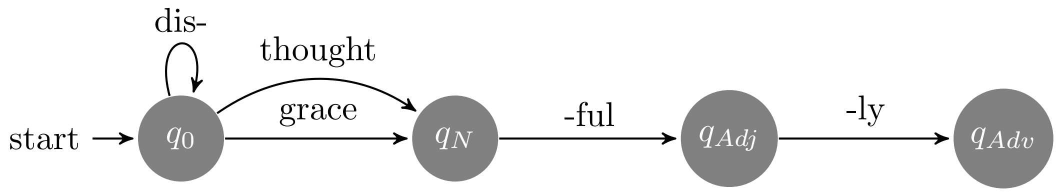


Figure 8.2: Second try at modeling English morphology, this time distinguishing parts-of-speech

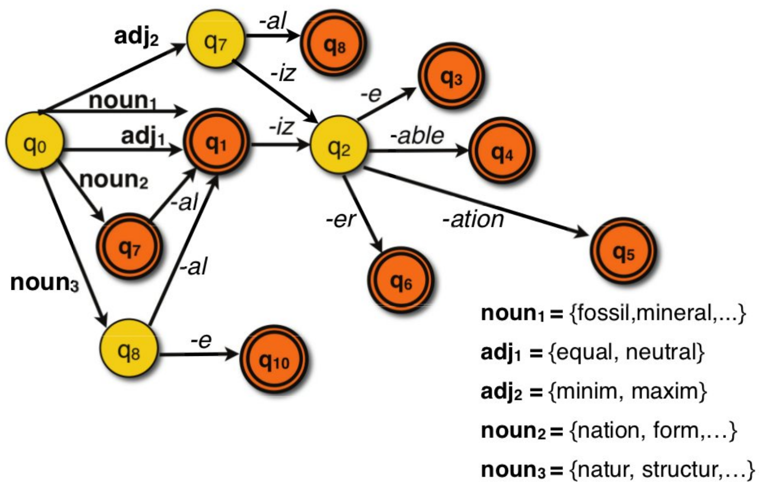


Figure 8.3: A fragment of a finite-state acceptor for derivational morphology. From Julia Hockenmaier's slides.

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 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.

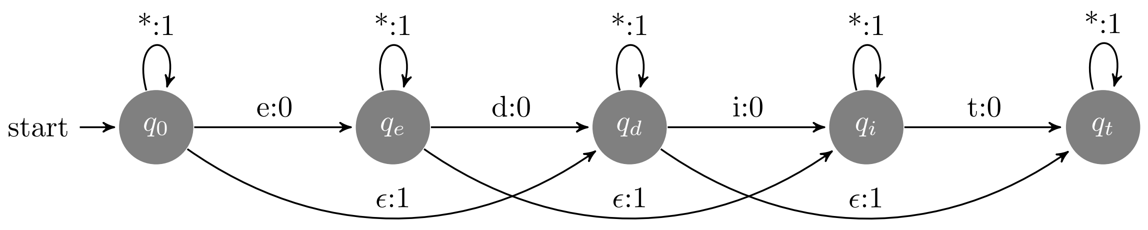


Figure 8.4: A weighted finite state acceptor for computing edit distance from the word *edit*.

8.2.1 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 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, $\delta_{q_i,\omega_m,q_j} = \text{p}(q_i, \omega_m \mid q_j)$, so that the product is equal to the joint probability of the state sequence and the string,

$$\text{p}(\boldsymbol{q}_{1:M}, \boldsymbol{\omega}_{1:M}) = \prod_{m}^{M} \text{p}(q_m, \omega_m \mid q_{m-1}). \tag{8.5}$$

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, $\delta_{q_0,w,q_0} = \text{p}_1(w)$.

To construct a bigram language model, we need to model the conditional probability $\text{p}(w_m \mid w_{m-1})$. To do this in a WFSA, we must create V different states: one for each context. Then we define the transition function as,

$$\delta_{q_i,w_m,q_j} = \begin{cases} \text{p}(w_m \mid w_{m-1} = i), & j = m \\ 0, & \text{otherwise.} \end{cases} \tag{8.6}$$

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 $\text{p}(w \mid \lozenge)$, and the final state function captures the probability $\text{p}(\blacklozenge \mid w)$. Thus, the bigram probability of any string is computed by the product of transition scores,

$$\text{p}_2(\boldsymbol{w}_{1:M}) = \text{p}(w_1 \mid \lozenge) \times \left(\prod_{m=2}^{M} \text{p}(w_m \mid w_{m-1}) \right) \times \text{p}(\blacklozenge \mid w_M) \tag{8.7}$$

$$= \pi_{w_1} \times \left(\prod_{m=2}^{M} \delta_{q_{w_{m-1}},w_m,q_{w_m}} \right) \times \xi_{w_M}. \tag{8.8}$$

Can you see how to construct a trigram language model in the same way?

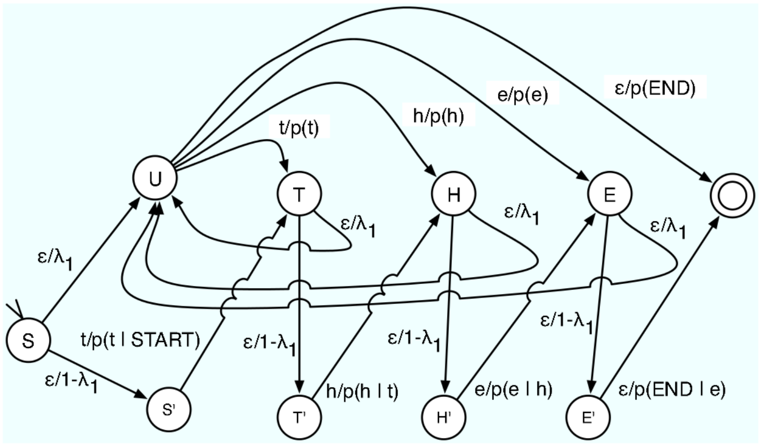


Figure 8.5: WFSA implementing an interpolated bigram/unigram language model (Knight and May, 2009). [todo: maybe redraw this for clarity?]

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,

$$\hat{\mathrm{p}}(w_m \mid w_{m-1}) = \lambda \mathrm{p}_1(w_m) + (1-\lambda)\mathrm{p}_2(w_m \mid w_{m-1}), \tag{8.9}$$

with $\hat{\mathrm{p}}$ 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.

8.3.1 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.

$$\begin{aligned} score(\{a,b,a\}) = \bar{1} &\otimes \big(\lambda \otimes \mathrm{p}_2(a|*) \oplus (1-\lambda) \otimes \mathrm{p}_1(a)\big) \\ &\otimes \big(\lambda \otimes \mathrm{p}_2(b|a) \oplus (1-\lambda) \otimes \mathrm{p}_1(b)\big) \\ &\otimes \big(\lambda \otimes \mathrm{p}_2(a|b) \oplus (1-\lambda) \otimes \mathrm{p}_1(a)\big) \end{aligned}$$

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:

$$\begin{aligned} score(\{a,b,a\}) &= 0 + \log\big(\exp(\log\lambda + \log \mathrm{p}_2(a|*)) + \exp(\log(1-\lambda) + \log \mathrm{p}_1(a))\big) \\ &\quad + \log\big(\exp(\log\lambda + \log \mathrm{p}_2(b|a)) + \exp(\log(1-\lambda) + \log \mathrm{p}_1(b))\big) \\ &\quad + \log\big(\exp(\log\lambda + \log \mathrm{p}_2(a|b)) + \exp(\log(1-\lambda) + \log \mathrm{p}_1(a))\big) \\ &= 0 + \log(\lambda \mathrm{p}_2(a|*) + (1-\lambda)\mathrm{p}_1(a)) \\ &\quad + \log(\lambda \mathrm{p}_2(b|a) + (1-\lambda)\mathrm{p}_1(b)) \\ &\quad + \log(\lambda \mathrm{p}_2(a|b) + (1-\lambda)\mathrm{p}_1(a)), \end{aligned}$$

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:

$$Q = \{q_0, q_{\text{regular}}, q_{\text{needs-e}}, q_{\text{pluralized}}\} \tag{8.19}$$
$$N = \{\textit{aardvark}, \ldots, \textit{wish}, \textit{wit}, \ldots, \textit{zyzzyva}^2\}(\text{the set of all English nouns}) \tag{8.20}$$
$$\Sigma = N \cup \{+\text{PL}\} \tag{8.21}$$
$$\Delta = N \cup \{+s, +es\} \tag{8.22}$$
$$q_0 = q_0 \tag{8.23}$$
$$F = \{q_{\text{regular}}, q_{\text{needs-e}}, q_{\text{pluralized}}\} \tag{8.24}$$
$$\begin{aligned}\delta = \{ & (\langle q_0, \textit{aardvark} \rangle \rightarrow \langle q_{\text{regular}}, \textit{aardvark} \rangle), \\ & (\langle q_0, \textit{wish} \rangle \rightarrow \langle q_{\text{needs-e}}, \textit{wish} \rangle), \\ & (\langle q_0, \textit{wit} \rangle \rightarrow \langle q_{\text{regular}}, \textit{wit} \rangle), \\ & \ldots \\ & (\langle q_{\text{regular}}, +\text{PL} \rangle \rightarrow \langle q_{\text{pluralized}}, +s \rangle) \\ & (\langle q_{\text{needs-e}}, +\text{PL} \rangle \rightarrow \langle q_{\text{pluralized}}, +es \rangle)\end{aligned} \tag{8.25}$$

This machine will accept the pairs $\langle \textit{wit}+\text{PL}, \textit{wits} \rangle, \langle \textit{wish}+\text{PL}, \textit{wishes} \rangle, \langle \textit{wit}, \textit{wit} \rangle$, but not the pairs $\langle \textit{wit}+\text{PL}, \textit{wites} \rangle, \langle \textit{wish}+\text{PL}, \textit{wishs} \rangle, \langle \textit{wish}+\text{PL}, \textit{wish} \rangle$. 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)).

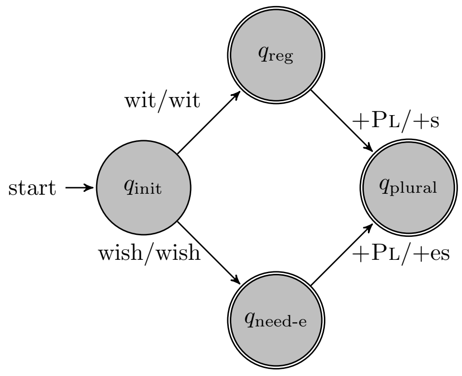


Figure 8.6: A finite state transducer for pluralizing English words.

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 § 8.2.1, 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.

- $Q_0 \overset{a}{\underset{a}{\to}} Q_0 : 0$
- $Q_0 \overset{a}{\underset{\epsilon}{\to}} Q_0 : 1$
- $Q_0 \overset{\epsilon}{\underset{a}{\to}} Q_0 : 1$

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 $\langle s, \epsilon \rangle, \langle h, h \rangle, \langle e, e \rangle$.

8.5.1 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.

8.5.2 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

8.6.1 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.

8.6.2 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 $\log \mathrm{p}_r(c_1, c_2, \ldots, c_M)$.
- T_2 is a character-to-character transliteration model. This can be based on explicit rules,[3] or on conditional probabilities $\log \mathrm{p}_t(c^{(f)} \mid c^{(r)})$.
- T_3 is an acceptor for a given string that is to be transliterated.

[3] `http://en.wikipedia.org/wiki/Romanization_of_Russian`

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,

8.6.3 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 $\log \mathrm{p}(w_1, w_2, \ldots, w_M)$. This model's responsibility is to tell us that $\mathrm{p}(\textit{Coffee black me pleases much}) \ll \mathrm{p}(\textit{I like black coffee a lot})$.
- 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 $\mathrm{p}(w^{(t)} \mid w^{(s)})$. This model should assign a high probability to $\mathrm{p}(\textit{cafe} \mid \textit{coffee})$, and a low probability to $\mathrm{p}(\textit{cafe} \mid \textit{tea})$.

  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 $\mathrm{p}(\boldsymbol{w}^{(e)})\mathrm{p}(\boldsymbol{w}^{(s)} \mid \boldsymbol{w}^{(e)}) = \mathrm{p}(\boldsymbol{w}^{(s)}, \boldsymbol{w}^{(e)})$.
- 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 $\mathrm{p}(\boldsymbol{w}^{(s)}, \boldsymbol{w}^{(e)})$. 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,

$$\hat{\boldsymbol{w}}^{(e)} = \operatorname{argmax} \boldsymbol{w}^{(e)} \mathrm{p}(\boldsymbol{w}^{(s)}, \boldsymbol{w}^{(e)}) \tag{8.27}$$

$$= \operatorname{argmax} \boldsymbol{w}^{(e)} \mathrm{p}(\boldsymbol{w}^{(e)} \mid \boldsymbol{w}^{(s)}), \tag{8.28}$$

  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 $(Q_i, \omega) \rightarrow Q_o$ 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 $\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{y})$.

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

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?

9.1 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:

(9.1) *The cat is fat.*

(9.2) *The cat that the dog chased is fat.*

(9.3) **The cat that the dog is fat.*

(9.4) *The cat that the dog that the monkey kissed chased is fat.*

(9.5) **The cat that the dog that the monkey chased is fat.*

Proof sketch:

- K is the set of grammatical english sentences.
  It excludes examples (9.3) and (9.5).
- 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?

9.1.1 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**.

9.1.2 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,

(9.6) *The **processor has** 10 million times fewer transistors on it than todays typical micro-processors, **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:

$$\begin{aligned}
\text{S} &\rightarrow \text{NN VP} \\
\text{VP} &\rightarrow \text{VP3S} \mid \text{VPN3S} \mid \ldots \\
\text{VP3S} &\rightarrow \text{VP3S, VP3S, } \textit{and} \text{ VP3S} \mid \text{VBZ} \mid \text{VBZ NP} \mid \ldots
\end{aligned}$$

9.2 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 $\langle N, \Sigma, R, S\rangle$:

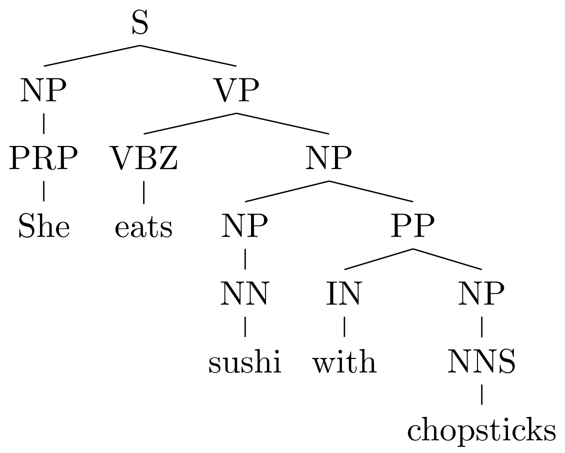
N	a set of non-terminals
Σ	a set of terminals (distinct from N)
R	a set of productions, each of the form $A \to \beta$, where $A \in N$ and $\beta \in (\Sigma \cup N)^*$
S	a designated start symbol

Context free grammars provide rules for generating strings.

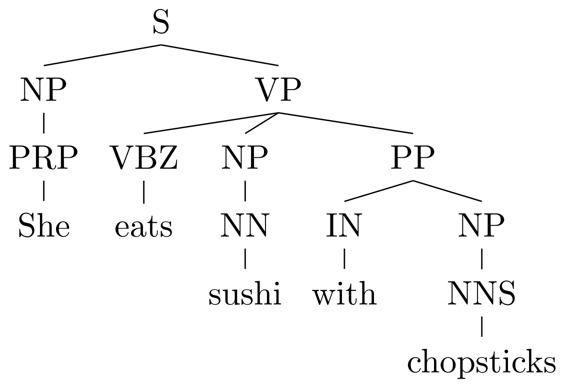
- The left-hand side (LHS) of each production is a non-terminal $\in N$
- The right-hand side (RHS) of each production is a sequence of terminals or non-terminals, $\{n, \sigma\}^*, n \in N, \sigma \in \Sigma$.

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 9.1.

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.



(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 9.1: Two derivations of the same sentence, shown as both parse trees and bracketings

Example The grammar below handles the case of center embedding:

$$\text{S} \to \text{NP VP}_1 \tag{9.1}$$
$$\text{NP} \to \textit{the}\ \text{NP} \ \mid \text{NP } \textsc{RelClause} \tag{9.2}$$
$$\textsc{RelClause} \to \textit{that}\ \text{NP V}_t \tag{9.3}$$
$$\text{V}_t \to \textit{ate} \mid \textit{chased} \mid \textit{befriended} \mid \ldots \tag{9.4}$$
$$\text{N} \to \textit{cat} \mid \textit{dog} \mid \textit{monkey} \mid \ldots \tag{9.5}$$
$$\text{VP}_1 \to \textit{is fat} \tag{9.6}$$

Here we are using a shorthand, where $\alpha \to \beta \mid \gamma$ implies two productions, $\alpha \to \beta$ and $\alpha \to \gamma$.

Semantics Ideally, each derivation will have a distinct semantic interpretation, and all possible interpretations will be represented in some derivation.

$$(_{\text{NP}}(_{\text{NP}}\ \textit{Ban}\ (_{\text{PP}}\ \textit{on}\ (_{\text{NP}}\ \textit{nude dancing}\ )))$$
$$(_{\text{PP}}\ \textit{on}\ (_{\text{NP}}\ \textit{Governor's desk}\ )))$$

$$(_{\text{NP}}\ \textit{Ban}\ (_{\text{PP}}\ \textit{on}\ (_{\text{NP}}(_{\text{NP}}\ \textit{nude dancing}\ )$$
$$(_{\text{PP}}\ \textit{on}\ (_{\text{NP}}\ \textit{Governor's desk}\ )))))$$

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*:

"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.

9.3 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 9.1 (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 9.1 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.

9.3.1 Substitution

Constituents generated by the same non-terminal should be substitutable in many contexts:

(9.7) ($_{\text{NP}}$ *The ban*) *is on the desk.*

(9.8) ($_{\text{NP}}$ *The Governor's desk*) *is on the desk.*

(9.9) ($_{\text{NP}}$ *The ban on dancing on the desk*) *is on the desk.*

(9.10) *($_{\text{PP}}$ *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.

(9.11) ($_{\text{NP}}$ *It*) *is on the desk.*

What about verbs?

(9.12) *I* ($_{\text{V}}$ *gave*) *it to Anne.*

(9.13) *I* ($_{\text{V}}$ *taught*) *it to Anne.*

(9.14) *I* ($_{\text{V}}$ *gave*) *Anne a fish*

(9.15) **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**. [todo: more details]

9.3.2 Coordination

Constituents generated by the same non-terminal can usually be *coordinated* using words like *and* and *or*:

(9.16) *We fought* ($_{\text{PP}}$ *on the hills*) *and* ($_{PP}$ *in the hedges*).

(9.17) *We fought* ($_{\text{ADVP}}$ *as well as we could*).

(9.18) **We fought* ($_{\text{ADVP}}$ *as well as we could*) *and* ($_{\text{PP}}$ *in the hedges*).

Like all such tests, coordination does not always work:

(9.19) *She* ($_{\text{VP}}$ *went*) ($_{\text{PP}}$ *to the store*).

(9.20) *She* ($_{\text{VP}}$ *came*) ($_{\text{PP}}$ *from the store*).

(9.21) *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 (9.22) *(The governor) banned (nude dancing on his desk)*

(9.23) *(Nude dancing on his desk) was banned by (the governor)*

Wh- movement (9.24) *(Nude dancing was banned) on (the desk).*

(9.25) *(The desk) is where (nude dancing was banned)*

Topicalization (9.26) *(He banned nude dancing) to appeal to conservatives.*

(9.27) *To appeal to conservatives, (he banned nude dancing).*

9.4 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.

9.4.1 Noun phrases

Let's start with noun phrases:

(9.28) ***She*** *sleeps* (Pronoun)

(9.29) ***Arlo*** *sleeps* (Proper noun)

These examples suggest that pronouns and proper nouns are substitutable, so we can define a production,

$$\text{NP} \rightarrow \text{Prp} \mid \text{Nnp}, \tag{9.7}$$

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,

$$\text{Prp} \rightarrow \textit{she} \mid \textit{he} \mid \textit{I} \mid \textit{you} \ldots \tag{9.8}$$

$$\text{Nnp} \rightarrow \textit{Arlo} \mid \textit{Abigail} \ldots \tag{9.9}$$

What else could be a noun phrase?

(9.30) ***A lobster*** *sleeps*

(9.31) ***The lobster*** *sleeps*

(9.32) ***Lobsters*** *sleep*

(9.33) ****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:

$$\text{NP} \rightarrow \text{Dt NOM} \mid \text{Nns} \tag{9.10}$$

$$\text{NOM} \rightarrow \text{Nn} \mid \text{Nns} \tag{9.11}$$

Notice that these productions also allow *The lobsters sleep*, using the NOM → Nns production.

Noun phrases may also contain various **modifiers**.

(9.34) ***The blue fish*** *sleeps* (adjective)

(9.35) ***The four crabs*** *sleep* (cardinality)

We could try to handle these cases by adding to the nominal productions,

$$\text{NOM} \rightarrow \text{JJ NOM} \mid \text{CD NOM} \tag{9.12}$$

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:

(9.36) *The girl from Omaha sleeps* (prepositional phrase)

(9.37) *Cats in Catalonia cry* (prepositional phrase)

(9.38) *The student who ate 15 donuts sleeps* (relative clause)

(9.39) *Mary from Omaha sleeps*

(9.40) *Cats who are in Catalonia cry*

(9.41) ?*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 (9.41), then we can handle both of these cases by adding the following NP productions,

$$\text{NP} \rightarrow \text{NP PP} \mid \text{NP RELCLAUSE} \tag{9.13}$$

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,

$$\begin{aligned} \text{NP} \rightarrow & \text{PRP} \mid \text{NNP} \mid \text{DT NOM} \mid \text{NP PP} \mid \text{NP RELCLAUSE} \\ \text{NOM} \rightarrow & \text{NN} \mid \text{ADJP NOM} \mid \text{CD NNS} \mid \text{NNS} \end{aligned}$$

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.

9.4.2 Adjectival and prepositional phrases

The noun phrase grammar mentioned prepositional phrases, such as

(9.42) *cats* ***from Catalonia***

(9.43) *pizza* ***in the refigerator***

(9.44) *pizza* ***in the old, broken refigerator***

(9.45) *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 (9.45). This suggests the simple production,

$$\text{PP} \rightarrow \text{P NP}. \tag{9.14}$$

The noun phrase fragment also includes adjective modifiers, like *the blue lobster*. But in fact, adjectives can combine into phrases.

(9.46) *the* ***large blue*** *fish*

(9.47) *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:

$$\text{ADJP} \rightarrow \text{J{\scriptsize J}} \mid \text{R{\scriptsize B}}\ \text{ADJP} \mid \text{J{\scriptsize J}}\ \text{ADJP} \tag{9.15}$$

$$\text{NOM} \rightarrow \text{ADJP}\ \text{N{\scriptsize N}} \mid \text{ADJP}\ \text{N{\scriptsize NS}} \tag{9.16}$$

Notice that if we instead added NOM $\rightarrow$ 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.

9.4.3 Verb phrases

Let's now consider the verb and its modifiers.

(9.48) *She* ***sleeps***

(9.49) *She* ***sleeps restlessly***

(9.50) *She* ***sleeps at home***

(9.51) *She* ***eats sushi***

(9.52) *She* ***gives John sushi***

Each of these examples requires a production,

$$\text{VP} \rightarrow \text{V} \mid \text{VP Rb} \mid \text{VP PP} \mid \text{V NP} \mid \text{V NP NP} \tag{9.17}$$

But what about **She sleeps sushi* or **She speaks John Japanese*? We need a more fine-grained verb non-terminal to handle these cases.

$$\begin{aligned}
\text{VP} &\rightarrow \text{VP Rb} \mid \text{VP PP} && (9.18)\\
\text{VP} &\rightarrow \text{V-intrans} \mid \text{V-trans NP} \mid \text{V-ditrans NP NP} && (9.19)\\
\text{V-intrans} &\rightarrow \textit{sleeps} \mid \textit{talks} \mid \textit{eats} \mid \ldots && (9.20)\\
\text{V-trans} &\rightarrow \textit{eats} \mid \textit{knows} \mid \textit{gives} \mid \ldots && (9.21)\\
\text{V-ditrans} &\rightarrow \textit{gives} \mid \textit{tells} \mid \ldots && (9.22)
\end{aligned}$$

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.*

9.4.4 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,

(9.53) *She eats sushi*

To handle this we simply need,

$$\text{S} \rightarrow \text{NP VP}. \tag{9.23}$$

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:

(9.54) *Sometimes, she eats sushi*

(9.55) *In Japan, she eats sushi*

We therefore add two more productions,

$$\begin{aligned}
\text{S} &\rightarrow \text{AdvP S} && (9.24)\\
\text{S} &\rightarrow \text{PP S} && (9.25)
\end{aligned}$$

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.

9.4.5 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,

(9.56) *She eats (sushi) and (candy)*

(9.57) *She (eats sushi) and (drinks soda)*

(9.58) *(She eats sushi) and (he drinks soda)*

(9.59) *(fresh) and (tasty) sushi*

These examples motivate, respectively, the following productions,

$$\text{NP} \rightarrow \text{NP } \textsc{Cc} \text{ NP} \tag{9.26}$$
$$\text{VP} \rightarrow \text{VP } \textsc{Cc} \text{ VP} \tag{9.27}$$
$$\text{S} \rightarrow \text{S } \textsc{Cc} \text{ S} \tag{9.28}$$
$$\text{ADJP} \rightarrow \text{ADJP } \textsc{Cc} \text{ ADJP} \tag{9.29}$$
$$\textsc{Cc} \rightarrow \textit{and} \mid \textit{or} \mid \ldots \tag{9.30}$$

We would need a little more cleverness to properly cover coordinations of more than two elements.

9.4.6 Odds and ends

Consider the example,

(9.60) *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.

$$\textsc{RelClause} \rightarrow \textsc{Wp} \text{ VP} \tag{9.31}$$
$$\textsc{Wp} \rightarrow \textit{who} \mid \textit{that} \mid \textit{which} \mid \ldots \tag{9.32}$$

Here are some related examples:

(9.61) *I took sushi from the man **offering sushi**.*

(9.62) *I gave sushi to the woman **working at home**.*

This is a gerundive postmodifier, which again requires its own non-terminal.

$$\text{NOM} \rightarrow \text{NOM GERUNDVP} \tag{9.33}$$
$$\text{GERUNDVP} \rightarrow \text{VBG} \mid \text{VBG NP} \mid \text{VBG PP} \mid \ldots \tag{9.34}$$
$$\text{VBG} \rightarrow \textit{offering} \mid \textit{working} \mid \textit{talking} \mid \ldots \tag{9.35}$$

Finally, we need to deal with questions, such as ***can** she eat sushi?* (and notice it's not *can she **eats** sushi*).

$$\text{S} \rightarrow \text{AUX NP VP} \tag{9.36}$$
$$\text{AUX} \rightarrow \textit{can} \mid \textit{did} \mid \ldots \tag{9.37}$$

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 10.

9.5 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:

$$A \rightarrow BC$$
$$A \rightarrow a$$

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 the production,

$$\text{W} \rightarrow \text{X Y Z}, \tag{9.38}$$

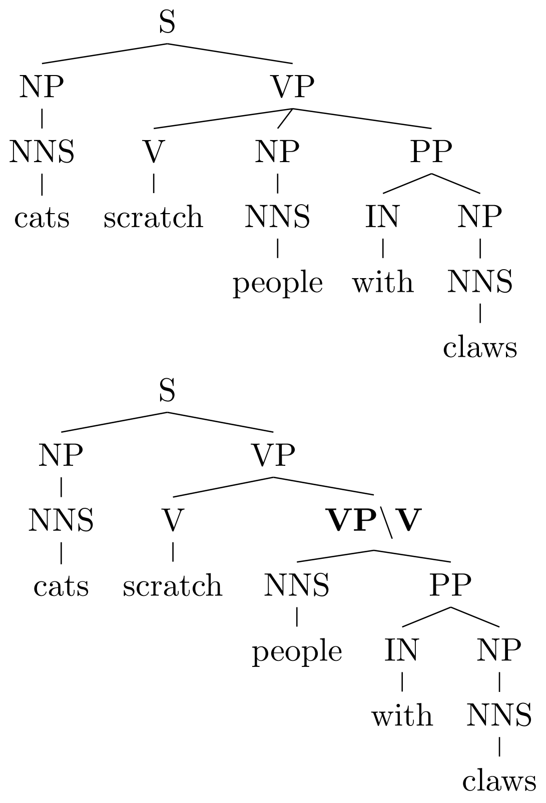


Figure 9.2: Binarization of the VP → V NP PP production

we can replace it with two productions,

$$W \rightarrow X\ W\backslash X \tag{9.39}$$
$$W\backslash X \rightarrow Y\ Z. \tag{9.40}$$

In these productions, W\X is a new dummy non-terminal, indicating a phrase that would be W if its left neighbor is an X. Figure 9.2 conveys this idea in a real example, with the non-terminal VP\V indicating a verb phrase that requires its left neighbor to be a verb.

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 10

Context-free Parsing

Parsing is the task of determining whether a string can be produced by a given context-free grammar, and if so, how. The "how" question involves obtaining a hierarchical structure, as shown in Figure 9.1 in the previous chapter. Before we discuss specific parsing algorithms, let us consider whether exhaustive search is possible. Suppose we only have one non-terminal, X, and it has the following productions:

$$\text{X} \rightarrow \text{X X}$$
$$\text{X} \rightarrow \textit{aardvark} \mid \textit{abacus} \mid \ldots \mid \textit{zyther}$$

In this grammar, the number of possible derivations for each string is equal to the number of binary bracketings, which is a **Catalan number**. Catalan numbers grow **super-exponentially** in the length of the sentence, $C_n = \frac{(2n)!}{(n+1)!n!}$. Clearly we cannot search the space of possible derivations naïvely; as with sequence labeling, we will make independence assumptions that allow us to search efficiently by reusing shared substructures with dynamic programming. This chapter will focus on a bottom-up algorithm called **CKY**; chapter 11 will describe a left-to-right algorithm called **shift-reduce**.

10.1 Deterministic bottom-up 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.

[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.

Let's start with a simple example grammar:

$$
\begin{aligned}
\text{S} &\rightarrow \text{NP VP} \\
\text{NP} &\rightarrow \text{NP PP} \mid \textit{we} \mid \textit{sushi} \mid \textit{chopsticks} \\
\text{PP} &\rightarrow \text{P NP} \\
\text{P} &\rightarrow \textit{with} \\
\text{VP} &\rightarrow \text{V NP} \mid \text{V PP} \\
\text{V} &\rightarrow \textit{eat}
\end{aligned}
$$

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 V 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 $\boldsymbol{w}_{i:j-1}$. The algorithm fills in the upper right triangle of the table; it begins with the diagonal, which corresponds to substrings of length 1, and the computes derivations for progressively larger substrings, until reaching the upper right corner $t[0, M]$, which corresponds to the entire input. If the start symbol S is in $t[0, M]$, then the string $\boldsymbol{w}$ is in the language defined by the grammar.

- 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 $t[2, 3] = \{\text{NP}\}$, and so on.
- Then we fill in the next diagonal, in which each cell corresponds to a subsequence of length two: $t[0, 2], t[1, 3], \ldots, t[M - 2, M]$. These are filled in by looking for binary productions capable of producing at least one entry in each of the cells corresponding to left and right children. For example, the cell $t[1, 3]$ includes VP because the grammar includes the production VP $\rightarrow$ V NP, and we have V $\in t[1, 2]$ and NP $\in t[2, 3]$.
- When we move to the next diagonal, there is an additional decision to make: where to split the left and right children. The cell $t[i, j]$ corresponds to the subsequence $\boldsymbol{w}_{i:j-1}$, and we must choose some **split point** $i < k < j$, so that $\boldsymbol{w}_{i:k-1}$ is the left child and $\boldsymbol{w}_{k:j-1}$ is the right child. We do this by looping over all possible k, and then looking for productions that generate elements in $t[i, k]$ and $t[k, j]$; the left-hand side of all such productions can be added to $t[i, j]$. When it is time to compute $t[i, j]$,

Algorithm 8 The CKY algorithm for parsing with context-free grammars

```
1: for $m \in \{0 \ldots M-1\}$ do
2:     $t[m, m+1] \leftarrow \{X : X \rightarrow w_m \in R\}$
3:     for $n \in \{m+1 \ldots M\}$ do
4:         $t[m, n] \leftarrow \varnothing$
5: for $\ell \in \{2 \ldots M\}$ do
6:     for $m \in \{0 \ldots M-\ell\}$ do
7:         for $k \in \{m+1 \ldots m+\ell-1\}$ do
8:             $t[m, m+\ell] \leftarrow t[m, m+\ell] \cup \{X : (X \rightarrow Y\ Z) \in R \wedge Y \in t[m, k] \wedge Z \in t[k, m+\ell]\}$
9: if S $\in t[0, M]$ then
10:     return True
11: else
12:     return False
```

the cells $t[i, k]$ and $t[k, j]$ have already been filled in, since these cells correspond to shorter sub-strings of the input.

- The process continues until we reach $t[0, M]$.

Algorithm 8 further formalizes this process, and Figure 10.1 shows the chart that arises from parsing the sentence *we eat sushi with chopsticks* using the grammar defined above.

An important detail about the CKY algorithm is that it assumes that all productions with non-terminals on the right-hand side (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 (see § 9.5). For example, if we have the production VP $\rightarrow$ V NP NP (for ditransitive verbs), we might convert to VP $\rightarrow$ VP$_{ditrans}$/NP NP, and then add the production VP$_{ditrans}$/NP $\rightarrow$ V NP.
- What about unary productions like VP $\rightarrow$ V? In practice, this is handled by making a second pass on each diagonal, in which each cell $t[i, j]$ is augmented with all possible unary productions capable of generating each item already in the cell. Suppose our example grammar is extended to include the production VP $\rightarrow$ V. Then the cell $t[1, 2]$ — corresponding to the word *eat* — would first include the set {V}, and would be augmented to the set {V, VP} during this second pass. This would then make it possible to parse sentences like *We eat.*

Computing the parse tree We are usually interested not only in whether a sentence is in a grammar, but in what syntactic structure is revealed by parsing. As with the Viterbi algorithm, we can compute this structure by keeping a set of back-pointers while populating the CKY table. If we add an entry X to cell $t[i, j]$ by using the production $X \rightarrow Y Z$ and

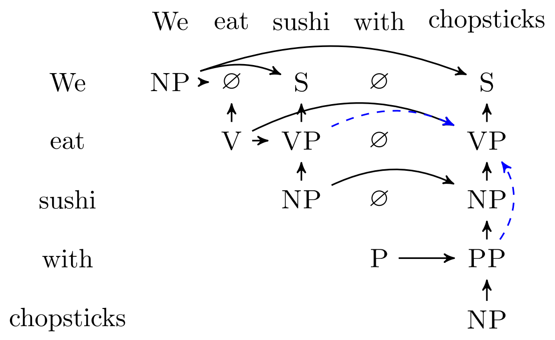


Figure 10.1: An example completed CKY chart. There are two paths to VP in position $t[1, 5]$, one in solid black and another in dashed blue.

the split point k, then we keep back-pointers $(i, j, X) \to (i, k, Y)$ and $(i, j, X) \to (k, j, Z)$. Once the table is constructed, we select back-pointers from $(0, M, \text{S})$, and recursively follow them until they ground out at individual words.

For ambiguous sentences, there will be multiple paths to reach S $\in t[0, M]$. For example, in Figure 10.1, we reach $t[0, M]$ through a production that includes VP$int[1, 5]$. Parsing is ambiguous for this example because there are two different ways to reach VP $\in t[1, 5]$: one with (*eat sushi*) and (*with chopsticks*) as children, and another with (*eat*) and (*sushi with chopsticks*) as children. The presence of multiple paths indicates that the input could have been generated by the grammar in more than one way. In § 10.2, we consider different ways to resolve this ambiguity.

Complexity The space complexity of the CKY algorithm 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. 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 $\mathcal{O}(M \log M)$.

10.2 Ambiguity in parsing

Unfortunately, parsing ambiguity is endemic to natural language:

- **Attachment ambiguity**: *we eat sushi with chopsticks, I shot an elephant in my pajamas.* In each of these examples, the prepositions (*with*, *in*) can attach to either the verb or the direct object.

- **Modifier scope**: *southern food store*, *plastic cup holder*. In these examples, the first word could be modifying the subsequent adjective, or the final noun.
- **Particle versus preposition**: *The puppy tore up the staircase.* Phrasal verbs like *tore up* often include particles which could also act as prepositions.
- **Complement structure**: *The tourists objected to the guide that they couldn't hear.* This is another form of attachment ambiguity, where the complement *that they couldn't hear* could attach to the main verb (*objected*), or to the indirect object (*the guide*).
- **Coordination scope**: *"I see," said the blind man, as he picked up the hammer and saw.* In this example, the lexical ambiguity for *saw* enables it to be coordinated either with the noun *hammer* or the verb *picked up*.

These forms of ambiguity can combine, so that a seemingly simple headlines like *Fed raises interest rates* can have dozens of possible analyses, even in a minimal grammar. Broad coverage grammars permit millions of parses of typical sentences. Faced with this ambiguity, classical deterministic 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.

Rather than attempting to design a grammar that achieves broad coverage and low ambiguity, contemporary methods use labeled data to learn models capable of selecting the correct syntactic analysis.

10.2.1 Parser evaluation

Before continuing to parsing algorithms that are able to handle ambiguity, 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.[2] But we would like to assign *partial credit* for correctly matching parts of the reference parse. The PARSEval metrics (Grishman et al., 1992) do that, scoring each system parse via:

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.

[2]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.

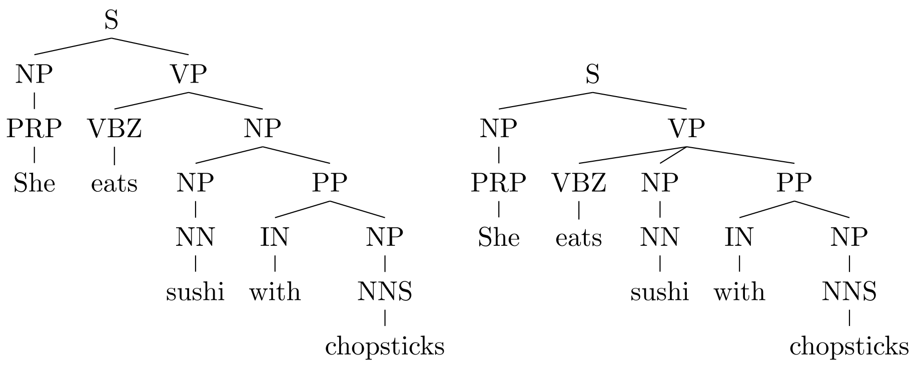


Figure 10.2: Suppose that the left parse is the system output, and the right parse is the ground truth; the precision is 0.75 and the recall is 1.0.

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 10.2, suppose that the left tree is the system parse and the right tree is the reference parse. We have the following spans:

- S $\to \boldsymbol{w}_{0:5}$ is **true positive**, because it appears in both trees.
- VP $\to \boldsymbol{w}_{1:5}$ is **true positive** as well.
- NP $\to \boldsymbol{w}_{2:5}$ is **false positive**, because it appears only in the system output.
- PP $\to \boldsymbol{w}_{3:5}$ is **true positive**, because it appears in both trees.

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.

10.2.2 Local solutions

Some ambiguity can be resolved locally. Consider the following examples,

(10.1) [*imposed* [*a ban* [*on asbestos*]]]

(10.2) [*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:

$$LR(v, n, p) = \frac{\text{p}(p \mid v)}{\text{p}(p \mid n)} = \frac{\text{p}(\textit{on} \mid \textit{imposed})}{\text{p}(\textit{on} \mid \textit{ban})} \tag{10.1}$$

where they select VERB attachment if $LR(v, n, p) > 1$. The probabilities are estimated from annotated training data.

This approach is capable of modeling which prepositions tend to attach to which verbs and nouns. However, it ignores any information about the object of the prepositional phrase, which might also factor into this decision. For example:

(10.3) ...[*it* [*would end* [*its venture* [*with Maserati*]]]]

(10.4) ...[*it* [*would end* [*its venture*][*with Maserati*]]]

The first analysis attaches *with Maserati* to the *venture*, which is almost surely preferred. Yet the likelihood ratio test ignores *Maserati*, and prefers to link the preposition *with* to the verb *end* (as in *It will end with a bang*).

- $\text{p}(\textit{with} \mid \textit{end}) = \frac{607}{5156} = 0.118$
- $\text{p}(\textit{with} \mid \textit{venture}) = \frac{155}{1442} = 0.107$

A richer probabilistic approach is undertaken by Collins and Brooks (1995), who model attachment as depending on four **heads**:

- the preposition (e.g., *with*)
- the VP attachment site (e.g., *end*)
- the NP attachment site (e.g., *venture*)
- the NP to be attached (e.g., *Maserati*)

They propose a backoff-based approach:

- First, look for counts of the tuple ⟨*with, Maserati, end, venture*⟩, and see how often VP and NP attachment were preferred for this tuple.
- If there are no counts for the full tuple, back off to the triples, ⟨*with, Maserati, end*⟩+⟨*with,end,venture*⟩+ ⟨*with,Maserati, venture*⟩, and count how often VP and NP attachment were preferred in each case.
- If there are no counts even for these triples, then try ⟨*with, Maserati*⟩+⟨*with,end*⟩+⟨*with,venture*⟩.
- Finally, if there are no counts for even these tuples, simply compute how often the preposition preferred NP or VP attachment. Since prepositions are a closed class, we can expect to have sufficient data for each preposition.

Accuracy of this method is roughly 84%. This approach of combining relative frequency estimation, smoothing, and backoff was very characteristic of 1990s statistical natural language processing. More conventional classification-based techniques can also be used for this problem; for example, Ratnaparkhi et al. (1994) designed a set of features and then trained a logistic regression classifier.

10.2.3 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 garden-path sentence,

(10.5) *Cats scratch people with claws with knives.*

We may want to attach *with claws* to *scratch*, as would be correct in the shorter 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.

10.3 Weighted Context-Free Grammars

In a **weighted context-free grammar** (WCFG), each production $X \to \alpha$ is associated with a score $\psi_{X \to \alpha}$. The score of a derivation is simply the combination (sum or product) of the scores of all the productions. For any given string $\boldsymbol{w}$, the "best" parse is the one corresponding to the highest-scoring derivation. More formally, for a given sequence $\boldsymbol{w}$, we want to select the parse τ that maximizes the score,

$$\hat{\tau} = \operatorname*{argmax}_{\tau:\text{yield}(\tau)=\boldsymbol{w}} \sum_{(X \to \alpha) \in \tau} \psi_{X \to \alpha},$$

where we model a derivation τ as a set of productions of the form $X \to \alpha$. 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.

10.3.1 Probabilistic context-free grammars

An important special case of WCFGs is a **probabilistic context-free grammar** (PCFG), in which the weight for each production $X \to \alpha$ corresponds to a log-probability, $\psi_{X \to \alpha} = \log \text{p}(\alpha \mid X)$. 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 $\to$V	0.3
VP $\to$V NP	0.6
VP $\to$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.

S	$\rightarrow$ NP VP	0.9
S	$\rightarrow$ S CC S	0.1
NP	$\rightarrow$ N	0.2
NP	$\rightarrow$ DT N	0.3
NP	$\rightarrow$ N NP	0.2
NP	$\rightarrow$ JJ NP	0.2
NP	$\rightarrow$ NP PP	0.1
VP	$\rightarrow$ V	0.4
VP	$\rightarrow$ V NP	0.3
VP	$\rightarrow$ V NP NP	0.1
VP	$\rightarrow$ VP PP	0.2
PP	$\rightarrow$ P NP	1.0

Table 10.1: A fragment of an example probabilistic context-free grammar (PCFG)

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 10.1 and the parse in Figure 10.3. The probability of this parse is:[3]

$$\begin{aligned}
\text{p}(\tau, \boldsymbol{w}) =& \text{p}(\text{S} \rightarrow \text{NP VP}) \\
& \times \text{p}(\text{NP} \rightarrow \text{N}) \times \text{p}(\text{N} \rightarrow \textit{they}) \\
& \times \text{p}(\text{VP} \rightarrow \text{VP PP}) \\
& \times \text{p}(\text{VP} \rightarrow \text{V NP}) \times \text{p}(\text{V} \rightarrow \textit{eat}) \\
& \times \text{p}(\text{NP} \rightarrow \text{N}) \times \text{p}(\text{N} \rightarrow \textit{sushi}) \\
& \times \text{p}(\text{PP} \rightarrow \text{P NP}) \times \text{p}(\text{P} \rightarrow \textit{with}) \\
& \times \text{p}(\text{NP} \rightarrow \text{N}) \times \text{p}(\text{N} \rightarrow \textit{chopsticks}) && (10.2) \\
=& 0.9 \times 0.2 \times 0.2 \times 0.3 \times 0.2 \times 1.0 \times 0.2 \\
& \times \text{probability of terminal productions} && (10.3)
\end{aligned}$$

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 $\rightarrow$ VP PP, we would have the production NP $\rightarrow$ NP PP. Since p(VP $\rightarrow$ VP PP) $>$ p(NP $\rightarrow$ NP PP) in the PCFG, the verb phrase attachment would be preferred.[4]

[3] In the remainder of the chapter, we will use the notation p$(X \rightarrow Y Z)$ for the probability of producing Y and Z, conditioned on the left-hand side being X.

[4] 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.

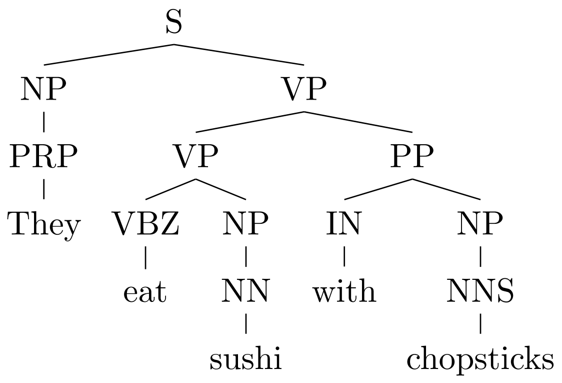


Figure 10.3: An example derivation

10.3.2 Feature-based parsing

The scores for each production can also be computed as an inner product of weights and features,

$$\psi_{X\to\alpha} = \boldsymbol{\theta}\cdot\boldsymbol{f}(X,\alpha), \tag{10.4}$$

where the feature vector $\boldsymbol{f}(X,\alpha,\boldsymbol{w})$ is a function of the left-hand side X and the right-hand side α. More generally, we can estimate weights for productions covering specific parts of the input,

$$\psi_{X\to\alpha,i,j,k} = \boldsymbol{\theta}\cdot\boldsymbol{f}(X,\alpha,\boldsymbol{w},i,j,k), \tag{10.5}$$

where the feature vector is now a function of the details of the production (X and α), as well as the text $\boldsymbol{w}$ and the indices of the spans to derive: the parent $\boldsymbol{w}_{i:j-1}$, the left child $\boldsymbol{w}_{i:k-1}$, and the right child $\boldsymbol{w}_{k:j-1}$. This is equivalent to characterizing the entire parse τ in terms of a locally-decomposable feature vector,

$$\boldsymbol{f}(\tau,\boldsymbol{w}) = \sum_{(X\to\alpha,i,j,k)\in\tau}\boldsymbol{f}(X,\alpha,\boldsymbol{w},i,j,k) \tag{10.6}$$

$$\hat{\tau} = \operatorname*{argmax}_{\tau}\boldsymbol{\theta}\cdot\boldsymbol{f}(\tau,\boldsymbol{w}) \tag{10.7}$$

$$= \operatorname*{argmax}_{\tau}\sum_{(X\to\alpha,i,j,k)\in\tau}\boldsymbol{\theta}\cdot\boldsymbol{f}(X,\alpha,\boldsymbol{w},i,j,k). \tag{10.8}$$

This enables the use of richer features, such as the words that border the span $\boldsymbol{w}_{i:j-1}$, the specific word at the split point w_k, the presence of a verb or noun in the left child span $w_{i:j-1}$, etc. The use of such features does not affect the applicability of the CKY parsing algorithm: we can still compute each element of the table $t[i,j]$ recursively, and therefore we can still find the best parse in polynomial time. The only restriction is that the features for each production $X \to \alpha$ cannot consider other non-terminals besides the parent X

and the children α. This is analogous to the Viterbi restriction that features consider only adjacent tags.

10.3.3 Estimation

Probabilistic context free grammars are similar to hidden Markov models, in that they are generative models of text. The parameters in hidden Markov models can be estimated from labeled data by relative frequency, and the same approach can be applied in PCFGs. In this case, the parameters of interest correspond to probabilities of productions, conditional on the left hand side. The relative frequency estimate is therefore,

$$\mathrm{p}(X \to \alpha) = \frac{\text{count}(X \to \alpha)}{\text{count}(X)}. \tag{10.9}$$

For example, the probability of the production NP $\to$ DT NN is equal to the count of this production divided by the count of the non-terminal NP. This applies to terminal productions as well: the probability of NN $\to$ *centipede* is the count of how often *centipede* appears in the corpus as generated from an NN tag, divided by the total count of the NN tag. These counts can be obtained from an annotated dataset, such as the **Penn Treebank**, which includes syntactic annotations over one million words of English text (Marcus et al., 1993). Even with one million words, it will be difficult to compute probabilities of relatively rare events, such as NN $\to$ *centipede*. Therefore, smoothing techniques will again be critical for making PCFGs effective.

Feature-based parsing models can be estimated using either structure perceptron or maximum conditional likelihood. For structure perceptron, we would compute,

$$\hat{\tau} = \underset{\tau:\text{yield}(\tau)=\boldsymbol{w}^{(i)}}{\operatorname{argmax}} \boldsymbol{\theta} \cdot \boldsymbol{f}(\tau, \boldsymbol{w}^{(i)}) \tag{10.10}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{f}(\tau^{(i)}, \boldsymbol{w}^{(i)}) - \boldsymbol{f}(\hat{\tau}, \boldsymbol{w}^{(i)}). \tag{10.11}$$

Alternatively, we can estimate the weights $\boldsymbol{\theta}$ by maximizing the conditional log-likelihood, $\sum_{i=1}^{N} \log \mathrm{p}(\tau^{(i)} \mid \boldsymbol{w}^{(i)})$, which is analogous to the conditional random field (CRF) model for sequence labeling. Finkel et al. (2008) present a CRF-based parsing model. Carreras et al. (2008) present a structure perceptron model, although they perform parsing in alternative syntactic formalism known as **Tree-Adjoining Grammar** (Joshi and Schabes, 1997). § 10.5.2 gives more details on these discriminative parsing models.

10.3.4 Parsing with weighted context-free grammars

It is not difficult to extend the CKY algorithm to include probabilities or other weights. Let us write $\psi_{X \to Y\ Z}$ for the score for the production $X \to Y\ Z$. In the original CKY algorithm for deterministic parsing, each cell $t[i, j]$ stored a set of non-terminals capable of deriving

Algorithm 9 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
                $t[m, m+\ell, X] \leftarrow t[m, m+\ell, X] + \max_{k, X \rightarrow Y\ Z} \psi_{X \rightarrow Y\ Z} + t[m, k, Y] + t[k, m+\ell, Z]$
```

the span $\boldsymbol{w}_{i:j-1}$. We now augment the table to be indexed by the tuple (i, j, X), with $t[i, j, X]$ indicating the score of the best possible derivation of $\boldsymbol{w}_{i:j-1}$ from non-terminal X. Algorithm 9 shows how to perform CKY parsing using such a table.

We also keep the back-pointers corresponding to the best path to $t[i, j, X]$; the best-scoring derivation can be obtained by tracing these pointers from $t[0, M, \text{S}]$ back to each terminal, just as the best sequence of labels in the Viterbi algorithm can be computed by tracing pointers backwards from the end of the trellis. Note that we need only store back-pointers for the **best** path to $t[i, j, X]$; this follows from the locality assumption that the score for a parse is a combination of the scores of each production in the parse.

Semiring CKY As with hidden Markov models, we can generalize weighted CKY parsing using semiring notation. The basic recurrence becomes,

$$t[m, m+\ell, X] = t[m, m+\ell, X] \otimes \left(\bigoplus_{k, X \rightarrow Y\ Z} \psi_{X \rightarrow Y\ Z} \otimes t[m, k, Y] \otimes t[k, m+\ell, Z] \right). \quad (10.12)$$

This notation makes it possible to capture a number of different parsing algorithms compactly. If the scores ψ correspond to log-probabilities or feature-weight inner products, then $\otimes$ is addition, as in Algorithm 9. If ψ are probabilities, then $\otimes$ is multiplication. By setting $\oplus$ equal to the $\max$ operation, the CKY algorithm computes the score of the best parse for a given sentence.

If the scores ψ correspond to probabilities, we can set $\oplus$ to summation and $\otimes$ to multiplication. Then the value at $t[0, M]$ corresponds to the total probability of all derivations of the input. This is known as the **inside algorithm**, and it is the tree-structured version of the **forward algorithm** from § 6.4.3. The inside algorithm can be used for unsupervised or semi-supervised parsing (Pereira and Schabes, 1992).

Finally, if we set $\otimes$ to be the boolean "and" operation, and $\oplus$ to be the boolean "or" operation, then $t[0, M] = \text{True}$ if and only if there is at least one derivation of the input from the grammar. Thus, the semiring notation generalizes across the weighted and unweighted CKY algorithms.

10.4 Improving Parsing by Refined Non-terminals

PCFG parsing on the Penn Treebank dataset does not perform well: Johnson (1998) shows that a PCFG estimated from treebank production counts obtains an F-measure of only $F = 0.72$. There are several problems with the use of weighted context free grammars on the Penn Treebank dataset:

- One problem is that the context-free assumption is too strict: for example, the probability of the production NP $\rightarrow$ NP PP is much higher if the parent of the noun phrase is a verb phrase (indicating that the NP is a direct object) than if the parent is a sentence (indicating that the NP is the subject of the sentence). Accurately modeling this "vertical" context is essential for accurate parsing.
- Another problem is that the Penn Treebank non-terminals are simply too coarse: there are many kinds of noun phrases and verb phrases, and accurate parsing sometimes requires knowing the difference. As we have already seen, when faced with prepositional phrase attachment ambiguity, a weighted CFG will either always choose NP attachment (if $\psi_{\text{NP}\rightarrow\text{NP PP}} > \psi_{\text{VP}\rightarrow\text{VP PP}}$), or it will always choose VP attachment. To get more nuanced behavior, more fine-grained non-terminals are needed.
- More generally, accurate parsing requires some amount of **semantics** — understanding the meaning of the text to be parsed. Consider the example *cats scratch people with claws*: knowledge of about *cats*, *claws*, and scratching is necessary to correctly resolve the attachment ambiguity.
- As a more extreme case, consider the example shown in Figure 10.4. The analysis on the left is preferred because of the conjunction of similar entities *France* and *Italy*. But given the non-terminals shown in the analyses, there is no way to differentiate these two parses, since they include exactly the same productions.

In all cases, what is needed seems to be more precise non-terminals. One possibility would be to rethink the linguistics behind the Penn Treebank, and ask the annotators to try again. But the original annotation effort took five years, and a more fine-grained set of non-terminals would only make things worse. We will therefore focus on automated techniques.

10.4.1 Parent annotations and other tree transformations

The key assumption underlying weighted context-free parsing is that productions depend only on the identity of the non-terminal on the left-hand side, and not on its ancestors

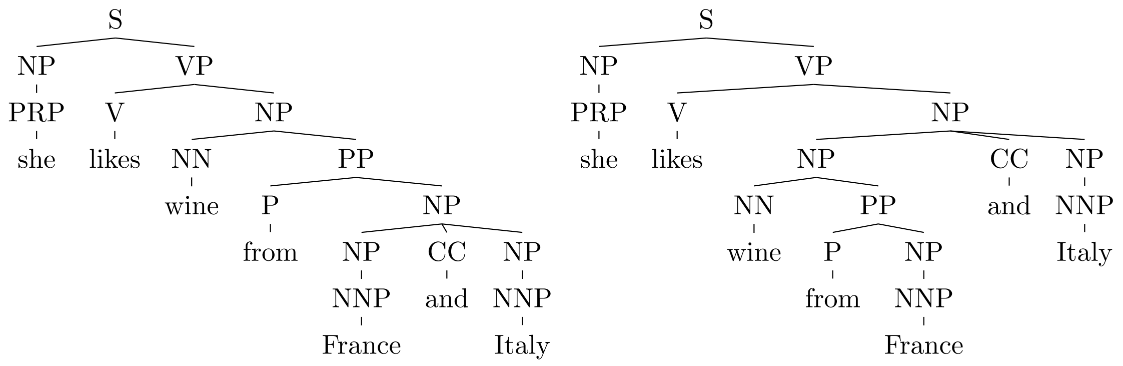


Figure 10.4: The left parse is preferable because of the conjunction of phrases headed by *France* and *Italy*.

in the parse. The validity of this assumption is an empirical question, and it depends on the non-terminals themselves: ideally, every noun phrase would be distributionally identical, so the assumption would hold. But in PTB-style analysis of English grammar, the observed probability of productions often depends on the parent of the left-hand side. For example, noun phrases are 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*). This means that the NP → NP PP production is more likely if the entire constituent is the child of a VP than if it is the child of S.

$$P(\text{NP} \rightarrow \text{NP PP}) = 11\% \tag{10.13}$$
$$P(\text{NP } \textsc{under } \text{S} \rightarrow \text{NP PP}) = 9\% \tag{10.14}$$
$$P(\text{NP } \textsc{under } \text{VP} \rightarrow \text{NP PP}) = 23\%. \tag{10.15}$$

Johnson (1998) proposes to capture this phenomenon via **parent annotation**. Each non-terminal is augmented with the identity of its parent, as shown in Figure 10.5). This is sometimes called **vertical Markovization**, since we introduce a Markov dependency between each node and its parent (Klein and Manning, 2003).

Using this transformation and a number of related heuristics, Johnson (1998) was able to improve the accuracy of PCFG-based parsing from 72% to 80%, at the cost of increasing the number of production rules from 14,962 to 22,773. (Recall that the number of production rules is a constant factor in the time complexity of WCFG parsing.) This increase in the number of rules is relatively modest, considering that parent annotation squares the number of non-terminals.

Parent annotation weakens the PCFG independence assumptions. This could improves accuracy by enabling the parser to make more fine-grained distinctions, which

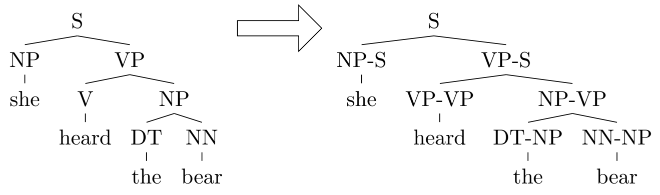


Figure 10.5: Parent annotation in a CFG derivation

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 10.2: A fragment of head percolation rules

better capture real lingusitic phenomena. However, each production is more rare (since the non-terminals are more specific), so the more careful smoothing is required to dampen the variance over production probabilities.

10.4.2 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. A fragment of the head percolation rules used in many parsing systems are found in Table 10.2.[5]

The meaning of the first rule 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 SBAR child, and so on down the list. Verb phrases are headed by left verbs (the head of *can walk home* is *walk*, since the modal verb *can* is tagged MD), noun phrases are headed by the rightmost noun-like non-terminal (so the head of *the red cat* is *cat*),[6] and prepositional phrases are headed

[5]From `http://www.cs.columbia.edu/~mcollins/papers/heads`

[6]The noun phrase non-terminal is sometimes treated as a special case. Collins (1997) uses a heuristic that

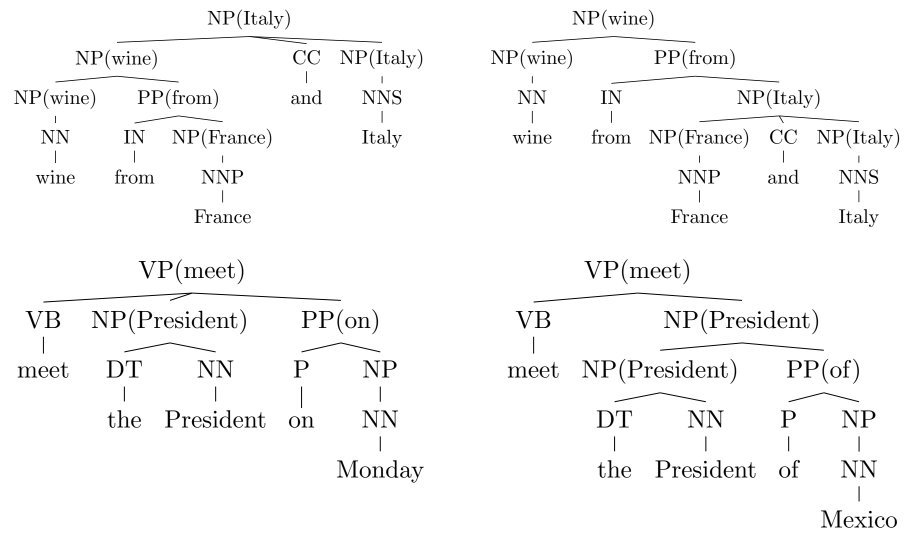


Figure 10.6: Lexicalization can address ambiguity on coordination scope (upper) and PP attachment (lower)

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 10.6.

- In the upper part of Figure 10.6, we see how lexicalization can help solve coordination scope ambiguity. We will correctly coordinate *France* and *Italy* if,

$$p(\text{NP}(\textit{Italy}) \to \text{NP}(\textit{France})\, \text{Cc}\, \text{NP}(\textit{Italy})) > p(\text{NP}(\textit{Italy}) \to \text{NP}(\textit{wine})\, \text{Cc}\, \text{NP}(\textit{Italy})). \tag{10.16}$$

- In the lower part of Figure 10.6, we see how lexicalization can help solve attachment ambiguity. Here we assume that,

$$p(\text{VP}(\textit{meet}) \to \alpha\, \text{PP}(\textit{on})) \gg p(\text{NP}(\textit{President}) \to \beta\, \text{PP}(\textit{on})) \tag{10.17}$$

$$p(\text{VP}(\textit{meet}) \to \alpha\, \text{PP}(\textit{of})) \ll p(\text{NP}(\textit{President}) \to \beta\, \text{PP}(\textit{of})) \tag{10.18}$$

In plain English: *meetings* are usually *on* things; *Presidents* are *of* things.

looks for the rightmost child which is a noun-like part-of-speech (e.g., *Nn, Nnp*), a possessive marker, or a superlative adjective (e.g., *the greatest*). If no such child is found, the heuristic then looks for the **leftmost** NP. If there is no child with tag NP, the heuristic then applies another priority list, this time from right to left.

- 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,

$$\text{p}(\text{VP} \to \text{V NP NP}) = 0.00151 \tag{10.19}$$

$$\text{p}(\text{VP}(\textit{said}) \to \text{V}(\textit{said}) \text{ NP NP}) = 0.00001 \tag{10.20}$$

$$\text{p}(\text{VP}(\textit{gave}) \to \text{V}(\textit{gave}) \text{ NP NP}) = 0.01980. \tag{10.21}$$

Overall, lexicalization had a major impact on parsing accuracy, which the best lexicalized parsers attaining accuracies in the range of 87-89% (Collins, 1997, 2003; Charniak, 1997). However, lexicalized parsing introduces significant technical challenges. First, the CKY parsing algorithm must keep track of the heads of each phrase, which adds algorithmic complexity. Second, the set of possible lexicalized productions is vastly larger, since it is quadratic in the size of the vocabulary (the left and right children each have a head, and one of these heads is chosen to head the unified phrase). We now briefly overview solutions to these problems.

Algorithms for lexicalized parsing

In weighted CFG parsing, the table element $t[i, j, X]$ keeps the score of the best derivation of the span $\boldsymbol{w}_{i:j-1}$ from the non-terminal X. However, for this constituent to participate on the right-hand side of higher-level productions, we must also know its head. One solution is to expand the table to include cells of the form $t[i, j, h, X]$, where h is the index of the head of the non-terminal X for the span $\boldsymbol{w}_{i:j-1}$, with $i \leq h < j$.

We can compute each element in the table by first computing the score of the best production in which the head comes from the left child, $t_\ell[i, j, h, X]$, then computing the score of the best production in which the head comes from the right child, $t_r[i, j, h, X]$, and finally taking the max over these two possibilities.

$$t_\ell[i, j, h, X] = \max_{X \to Y Z} \max_{k > h} \max_{k \leq h' < j} t[i, k, h, Y] + t[k, j, h', Z] + \psi_{X(h) \to Y(h) Z(h')} \tag{10.22}$$

$$t_r[i, j, h, X] = \max_{X \to Y Z} \max_{k \leq h} \max_{i \leq h' < j} t[i, k, h', Y] + t[k, j, h, Z] + \psi_{X(h) \to Y(h') Z(h)} \tag{10.23}$$

$$t[i, j, h, X] = \max\left(t_\ell[i, j, h, X], t_r[i, j, h, X]\right). \tag{10.24}$$

To compute t_ℓ, we maximize over all split points $k > h$, since the head word must be in the left child. We then maximize again over possible head words h' for the right child. An analogous computation is performed for t_r. The size of the table is now $\mathcal{O}(M^3 \#|N|)$, where M is the length of the input and $\#|N|$ is the number of non-terminals. Furthermore, each cell is computed by performing $\mathcal{O}(M^2)$ operations, since we maximize over both the split point k and the head h'. The time complexity of the algorithm is therefore $\mathcal{O}(M^5 \#|N|)$, which is impractical. Fortunately, the Eisner (1996) algorithm reduces this complexity back to $\mathcal{O}(M^3)$, using a more complex algorithm that maintains multiple tables.

The Charniak Parser

We now approach the problem of how to estimate weights for lexicalized productions $X(i) \rightarrow Y(j)\ Z(k)$. These productions are said to be **bilexical**, because they involve scores over pairs of words: in the example *... meet the President of Mexico*, we hope to choose the right attachment point by modeling the bilexical affinities of (*meet*, *of*) and (*President*, *of*). The number of such word pairs is of course quadratic in the size of the vocabulary, making it difficult to estimate them directly from data.

The Charniak (1997) parser addresses this issue in the context of probabilistic parsing, so that $\psi_{X(i) \rightarrow Y(j)\ Z(k)}$ is equal to the (log) probability of the lexicalized production. This probability is then decomposed into a product of: a **rule probability**, which is the probability of the unlexicalized production $X \rightarrow Y Z$, conditioned on the head word and parent of X (the same idea as parent annotation); a **head probability**, which is the probability of the head of X conditioned on X, the parent of X, and the head of the parent of X.

Recall the example from Figure 10.6, focusing on the bottom right example, *... meet the President of Mexico*. In the case of the production PP(*of*) $\rightarrow$ P(*of*) NP(*Mexico*), the rule probability is $\text{p}_{\text{rule}}(\text{PP} \rightarrow \text{P NP} \mid \text{PP}, \text{NP}, \textit{of})$, since the parent is a noun phrase and the head word is *of*. The head probability is $\text{p}_{\text{head}}(\textit{of} \mid \text{PP}, \text{NP}, \textit{President})$, since the parent is a noun phrase and the head of the parent is *President*. This captures the bilexical affinity between *President* and *of*, which is key to accurately parsing this example.

Even with this decomposition, it is necessary to smooth the rule and head probabilities to reduce the variance of the probability estimates. This is done by interpolating the full probabilities with simplified probabilities that condition on less information.

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, written $\blacklozenge$.

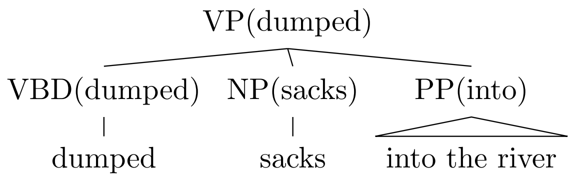


Figure 10.7: Example verb phrase for understanding the Collins parser

For example, consider the verb phrase *dumped sacks into the river*, shown in Figure 10.7. To model this rule, we would compute:

$$\mathrm{p}(\mathrm{VP}(\textit{dumped}, \mathrm{VBD}) \to [\blacklozenge, \mathrm{VBD}(\textit{dumped}, \mathrm{VBD}), \mathrm{NP}(\textit{sacks}, \mathrm{NNS}), \mathrm{PP}(\textit{into}, \mathrm{P}), \blacklozenge]),$$

with each phrase augmented by its head word and the head word's part of speech (e.g., VBD for the head word *dumped*).

This probability is computed through a generative model, in which the head is generated first (conditioned on the parent), and then each dependent is conditioned on the parent non-terminal and the head word. In this way, we do not directly estimate the full probability of a lexicalized production rule, but rather, we compute it from simpler probabilities involving the head and parent. Nonetheless, it is still necessary to smooth these probabilities by interpolating them with less expressive probability functions.

The Collins parser models bilexical dependencies between the head and its siblings. Bilexical probabilities require counts over pairs of words, a space of $\mathcal{O}(|\mathcal{V}|^2)$ events. It is this large event space that makes these probabilities difficult to estimate, necessitating smoothing. Is it worth it? Bikel (2004) evaluates the importance of bilexical probabilities to the performance of the Collins parser. He found that bilexical probabilites are rarely available — because most of the possible bilexical pairs in the test data are unobserved in the training data — but that bilexical probabilities are indeed active in 29% of the rules in the **top-scoring** parses. Still, 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.

10.4.3 Refinement grammars

Lexicalization improves on pure PCFG parsing by adding detailed information in the form of lexical heads. However, estimating the probabilities of lexicalized parsing rules is difficult, requiring additional independence assumptions and complex smoothing. Klein and Manning (2003) argue 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 10.8 shows an example of an error that is corrected through the introduction of a new NP-TMP subcategory for temporal noun phrases. Klein and Manning (2003) show how the introduction of a number of such categories can make unlexicalized PCFG parsing competitive with lexicalized methods.

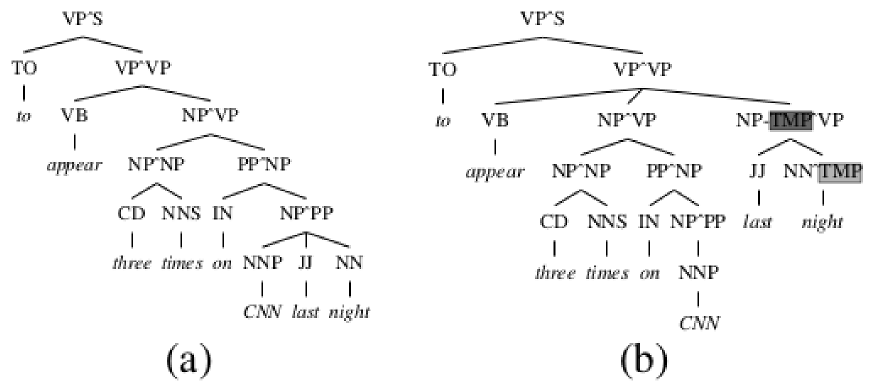


Figure 10.8: State-splitting creates a new non-terminal called NP-TMP, for temporal noun phrases. This corrects the PCFG parsing error in (a), resulting in the correct parse in (b).

***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 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 $\text{p}(X \rightsquigarrow \boldsymbol{w}_{i:j} \mid \boldsymbol{w}_{1:M})$, which is the probability that the span $\boldsymbol{w}_{i:j}$ is derived from the non-terminal X, conditioning on the entire sentence $\boldsymbol{w}_{1:M}$ and marginalizing over all

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 10.3: Examples of automatically refined non-terminals and some of the words that they generate (Petrov et al., 2006).

other parts of the derivation. Such marginals can be computed by a two-pass recursive algorithm called the **inside-outside algorithm** (Lari and Young, 1990).

- In the **inside** step, we compute the likelihood $\mathrm{p}(\boldsymbol{w}_{i:j} \mid X)$, which is simply the probability of deriving the span $\boldsymbol{w}_{i:j}$ from the non-terminal X; this probability depends only on the grammar, and not on any other parts of the sentence.
- In the **outside** step, we compute the probability $\mathrm{p}(X \mid \boldsymbol{w}_{1:i-1}, \boldsymbol{w}_{j+1:M})$, which is the probability of a non-terminal X governing the span $\boldsymbol{w}_{i:j}$, conditioned on the "outside" parts of the sentence, $\boldsymbol{w}_{1:i-1}$ and $\boldsymbol{w}_{j+1:M}$.

Each of these probabilities can be computed recursively. The marginal is then computed from the product of the inside and outside probabilities. The inside-outside algorithmic is a direct analogue of the forward-backward algorithm, which we used to compute the marginals necessary for training conditional random fields over sequence models.

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. Some examples of refined non-terminals are shown in Table 10.3. 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.

10.5 Discriminative parsing

The methods described in the previous section are all based on generative parsing models, in which the probability of a parse is a product of the probabilities of the individual productions. As we have seen, these models can be improved by using finer-grained non-terminals, via parent-annotation, lexicalization, and state-splitting. An alternative path to making parsing more accurate is to use techniques from discriminative machine learning. With the exception of reranking (discussed below), the introduction of discriminative methods to parsing came relatively late. The main reason is that these learning algorithms require multiple passes over the data, applying the parser repeatedly. Unlike sequence labeling, where the time complexity of inference is linear in the size of the input, the cost of inference for parsing is non-trivial — cubic in the length of the input. These limitations prevented well-known discriminative learning techniques, such as structured perceptron, from being applied sooner.

10.5.1 Reranking

An inexpensive way to get the benefits of discriminative learning is through **reranking** (Charniak and Johnson, 2005; Collins and Koo, 2005). First, a generative model — such as the Collins or Chariak parser — is used to identify the K-best parses for a sentence. (A modified version of CKY can compute the K-best parses efficiently.) Then a discriminative learning algorithm is trained to select the best of these parses. The discriminative model does not need to search over all parses, it only needs to consider the best K identified by the "generator." This means that the discriminator 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 rich lexicalized features, 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. The main limitation is that reranking can only find the best parse among the K-best offered by the generator, so it is inherently limited by the ability of the generator to find high-quality parse candidates.

10.5.2 Discriminative parsing

As shown in § 10.3.4, the weights on productions need not correspond to probabilities; the CKY algorithm can apply to **any** set of weights, as long as they are context-free. Discriminative learning can therefore be applied by setting $\psi_{X \to Y Z} = \boldsymbol{\theta} \cdot \boldsymbol{f}(X \to Y Z, \boldsymbol{w}, i, j, k)$, with the indices i, j, k indicating the boundaries of the parent ($\boldsymbol{w}_{i:j-1}$) and its left and right children ($\boldsymbol{w}_{i:k-1}$ and $\boldsymbol{w}_{k:j-1}$). Such features could incorporate lexical information, so that we learn weights for non-terminal productions as well as for lexicalized forms. For example:

- $f1$: 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 weighted CFG 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. Like CKY, the runtime of the inside-outside algorithmic is cubic in the length of the input. Because each instance must be visited and parsed many times during stochastic gradient descent, efficiency is critical. One solution is to "prefilter" the CKY parsing chart, identifying and eliminating 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; Carreras et al., 2008). They use features that capture "grandparent" dependencies between words and the heads of their parents' parents. These second-order dependency features make the time complexity $\mathcal{O}(M^4)$ in the length of the input, so pruning is again required to make parsing efficient enough to train accurately.

10.5.3 Neural parsing

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 **??**. [todo: say a little more more about durrett and klein]

Chapter 11

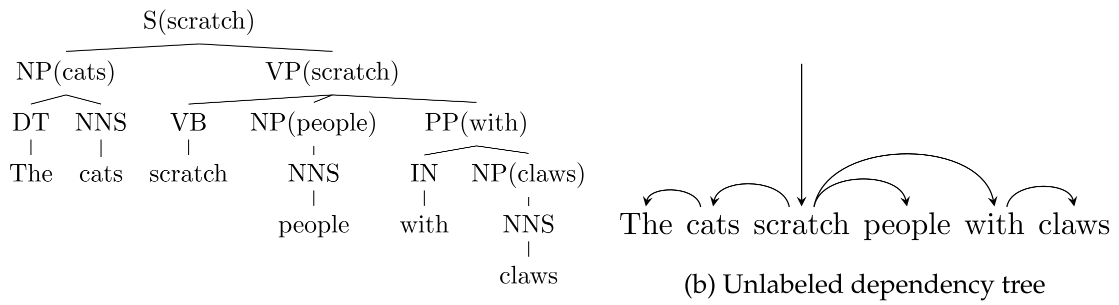
Dependency Parsing

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.

11.1 Dependency grammar

In lexicalized parsing, non-terminals such as NP are augmented with **head words**, as shown in Figure 11.1a. 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.

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* $\rightarrow$ *cats* and *cats* $\rightarrow$ *the*. We would not have the edge



(a) Lexicalized constituency parse (b) Unlabeled dependency tree

Figure 11.1: Dependency grammar is closely linked to lexicalized context free grammars: each lexical head has a dependency path to every other word in the constituent.

scratch $\rightarrow$ *the*, because although *scratch* dominates *the* in the graph, it 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 11.1b. 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.

11.1.1 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.

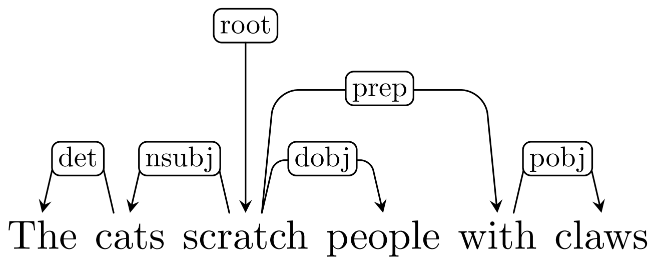


Figure 11.2: A labeled dependency parse

- The modifier may be optional while the head is mandatory: for example, in the sentence *cats scratch people with claws*, 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 to 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 11.2. 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.

11.1.2 Ambiguity and difficult cases

[todo: update this section with current standards from universal dependency treebank (Nivre et al., 2016)] The attachment ambiguity in the sentence shown in Figure 11.2 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 11.3 shows three different ways of representing prepositional phrase adjuncts to

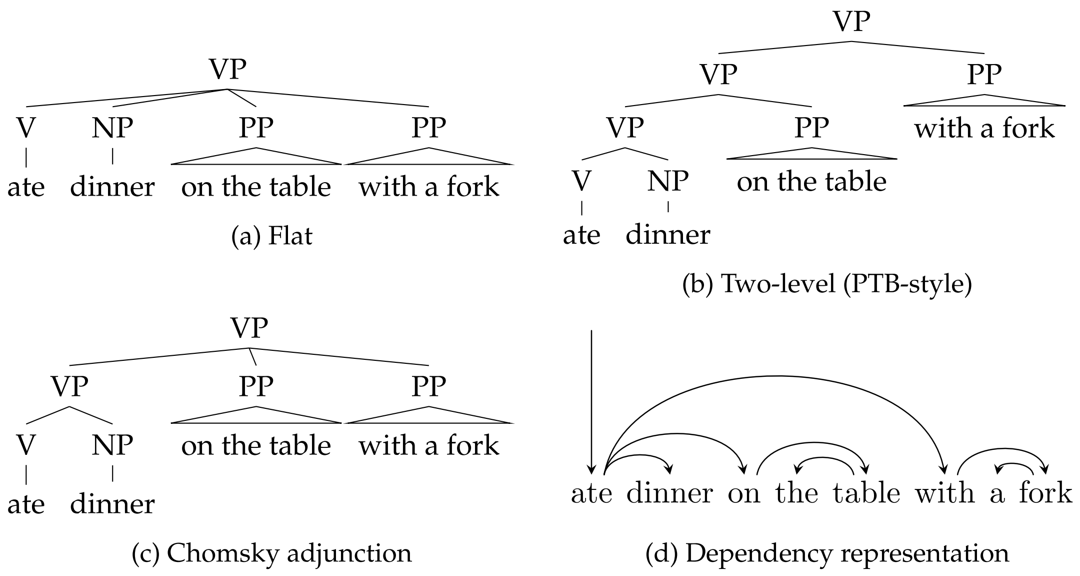
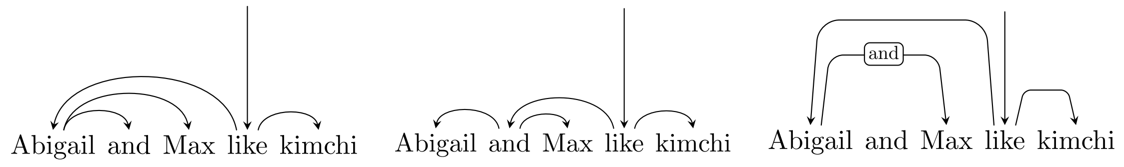


Figure 11.3: The three different CFG analyses of this verb phrase all correspond to a single dependency structure.

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 11.3d, 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 11.4), which word is the immediate dependent of the main verb *likes*? 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 11.4c, 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 § 10.4.2, the preposition


and
Abigail and Max like kimchi
Abigail and Max like kimchi
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 11.4: Three alternatives for representing coordination in a dependency parse

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*. Dependency annotation is an active area of research due to the ongoing development of the universal dependency treebank, which has produced dependency-annotated corpora in 47 languages at the time of this writing (Nivre et al., 2016).[1]

11.1.3 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 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 11.5 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

[1]`http://universaldependencies.org/`

	% non-projective edges	% non-projective sentences
Czech	1.86%	22.42%
English	0.39%	7.63%
German	2.33%	28.19%

Table 11.1: Frequency of non-projective dependencies in three languages (Kuhlmann and Nivre, 2010)

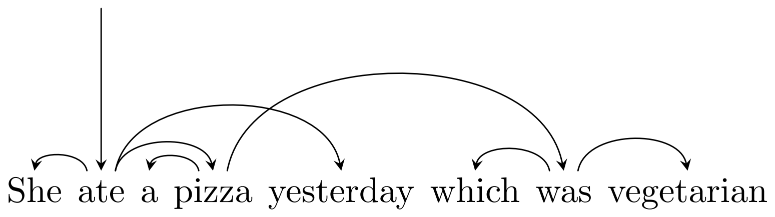


Figure 11.5: An example of a non-projective dependency parse in English

approximately 12,000 non-projective edges (see Table 11.1). 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.

11.2 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 $\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{y}, \boldsymbol{w})$, 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 prediction problem,

$$\hat{\boldsymbol{y}} = \underset{\boldsymbol{y} \in \mathcal{Y}(\boldsymbol{w})}{\operatorname{argmax}} \, \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{y}, \boldsymbol{w}). \tag{11.1}$$

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 $i \to j$:

$$\boldsymbol{f}(\boldsymbol{y}, \boldsymbol{w}) = \sum_{\langle i,j,r \rangle \in \boldsymbol{y}} \boldsymbol{f}(\boldsymbol{w}, i, j, r) \tag{11.2}$$

$$\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{y}, \boldsymbol{w}) = \sum_{\langle i,j,r \rangle \in \boldsymbol{y}} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, i, j, r). \tag{11.3}$$

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.

11.2.1 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 produced 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* $\to$ *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 include 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 § 11.2.5.

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* $\rightarrow$ *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."

11.2.2 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,

$$\hat{\boldsymbol{y}} = \operatorname*{argmax}_{\boldsymbol{y}' \in \mathcal{Y}(\boldsymbol{w})} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}') \tag{11.4}$$

$$\boldsymbol{\theta} = \boldsymbol{\theta} + \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}) - \boldsymbol{f}(\boldsymbol{w}, \hat{\boldsymbol{y}}) \tag{11.5}$$

This is just like sequence labeling, but now the argmax 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 $\mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w})$:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}} = \sum_{(i,j) \in \mathcal{Y}} \boldsymbol{f}(\boldsymbol{w}, i, j) - \sum_{i,j} \mathrm{p}(i \rightarrow j \mid \boldsymbol{w}) \boldsymbol{f}(\boldsymbol{w}, i, j) \tag{11.6}$$

Algorithm 10 Chu-Liu-Edmonds algorithm for unlabeled dependency parsing

```
1: procedure CHU-LIU-EDMONDS($\{\psi(i \to j)\}_{i,j \in \{1 \dots M\}}$)
2:     for $j \in 1 \dots M$ do
3:         $h_j \leftarrow \text{argmax}_i \, \psi(i \to j)$        ▷ Find the best incoming edge for each node
4:     $\tau \leftarrow \{j, h_j\}_{j \in 1 \dots M}$        ▷ $\tau$ is the graph of the best incoming edges
5:     $\mathcal{C} \leftarrow$ FINDCYCLES($\tau$)
6:     if $\mathcal{C} = \varnothing$ then
7:         return $\tau$        ▷ If $\tau$ has no cycles, it is the best tree
8:     else        ▷ Otherwise, collapse each cycle
9:         for each cycle $c \in \mathcal{C}$ do
10:            Remove all nodes in the cycle from the graph
11:            Add a "super-node" representing the cycle
12:        Let $G$ be the resulting graph
13:        return CHU-LIU-EDMONDS($G$)        ▷ Call recursively on the collapsed graph
```

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 also be computed in cubic time, using the **matrix-tree theorem** (Koo et al., 2007; McDonald et al., 2007; Smith and Smith, 2007). Details of these methods are described by Kübler et al. (2009).

11.2.3 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, $\psi(i \to j, r) = \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, i, j, r)$, 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.)

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 10, 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

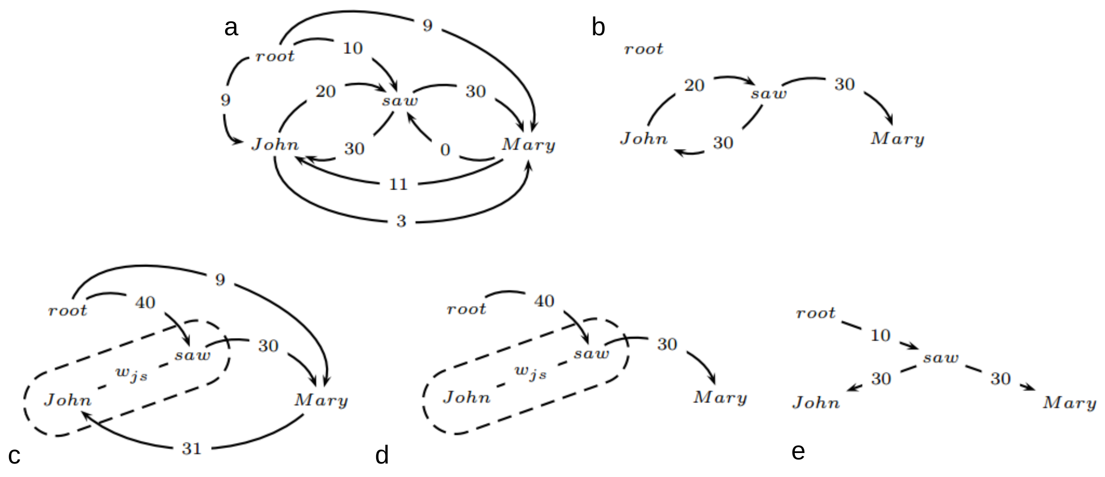


Figure 11.6: An illustration of the MST algorithm on a simple example. Figure borrowed from McDonald et al. (2005b).

the scores of the best spanning tree that includes both the edge and all nodes in the cycle. [todo: more detail on what happens when you collapse cycles].

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 original graph. The basic process is illustrated in Figure 11.6. 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 $\mathcal{O}(M^2 R)$.

11.2.4 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 § 10.4, 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:

$$t_\ell[i, j, h] = \max_{k>h} \max_{k \leq h' < j} t[i, k, h] + t[k, j, h'] + \psi(h \to h') \tag{11.7}$$

$$t_r[i, j, h] = \max_{k \leq h} \max_{i \leq h' < k} t[i, k, h'] + t[k, j, h] + \psi(h \to h') \tag{11.8}$$

$$t[i, j, h] = \max\left(t_\ell[i, j, h], t_r[i, j, h]\right). \tag{11.9}$$

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 11.7 and Equation 11.8. 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.[todo: write up formal algorithm description]

11.2.5 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 11.7. Second-order features consider **siblings** and **grandchildren**; third-order features consider **grand-siblings** (siblings and grandparents together) and **tri-siblings**.

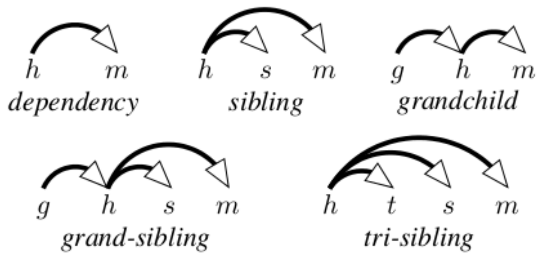


Figure 11.7: Feature templates for higher-order dependency parsing (Koo and Collins, 2010)

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 grandchild features, then our feature sets include:

- $\langle scratch \rightarrow with \rightarrow claws\rangle$
- $\langle people \rightarrow with \rightarrow claws\rangle$,

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).

11.3 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, because second-order dependency features are critical to correctly identify certain types of attachments. Graph-based dependency parsing may also be criticized 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, 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 the **arc-standard** parsing algorithm, which is similar to the LR algorithm that is used to parse programming languages. Transition-based parsing algorithms maintain a configuration state, which includes a stack where elements can be pushed and popped. They update the state incrementally through a series of actions, until the input is consumed and the stack is empty.

In the arc-standard parser, the configuration c is a tuple $c = (\sigma, \beta, A)$, where σ is a stack, β is the input buffer, and A is a set of dependency arcs. The initial state is $(\sigma = [0], \beta = \boldsymbol{w}_{1:M}, A = \varnothing)$, where: $\sigma = [0]$ indicates that the stack begins with the root node; $\beta = \boldsymbol{w}_{1:M}$ indicates that the buffer begins with the entire input string (indexed from 1); and $A = \varnothing$ means that there are not yet any arcs. We can then apply three possible actions:

- SHIFT: moves the first item from the input buffer on to the top of the stack,

$$(\sigma, i|\beta, A) \Rightarrow (\sigma|i, \beta, A), \tag{11.10}$$

  where we write $i|\beta$ to indicate that i is the leftmost item in the input buffer, and $\sigma|i$ to indicate the result of pushing i on to stack σ.

- ARC-LEFT: creates a new left-facing arc between the item on the top of the stack and the first item in the input buffer. This item is then "popped" to the front of the input buffer, and the arc is added to A.

$$(\sigma|i, j|\beta, A) \Rightarrow (\sigma, j|\beta, A \cup (j, \ell, i)), \tag{11.11}$$

  where ℓ is the (optional) label of the dependency arc.

- ARC-RIGHT: creates a new right-facing arc between the item on the top of the stack and the first item in the input buffer; this item is then "popped" to the front of the input buffer, and the arc is added to A.

$$(\sigma|i, j|\beta, A) \Rightarrow (\sigma, i|\beta, A \cup (i, \ell, j)), \tag{11.12}$$

  where again ℓ is the label of the dependency arc.

The ARC-LEFT action cannot be performed when the root node 0 is on top of the stack, since this node must be the root of the entire tree. Neither ARC-LEFT nor ARC-RIGHT can be performed if the result would create a second incoming edge for any word. When the stack σ and the input buffer β are empty, parsing is complete.

11.3.1 Learning transition-based parsers

Transition-based parsing requires selecting a series of actions. In parsing programming languages, shift-reduce parsers can choose the appropriate action deterministically, because programming languages are unambiguous by design. For natural language, we use machine learning classification to determine the best series of actions; for example, Yamada and Matsumoto (2003) use a support vector machine classifier (see § 2.1) to decide whether to shift or create a dependency arc at each stage in parsing.

To train a transition-based dependency parser, we can treat each parsing decision as a separate training instance. However, our ground truth input is not a list of parsing decisions, but rather, a dependency tree. We therefore require an **oracle** to convert the ground truth dependency tree into a list of parsing decisions, which can then be used as training data (Nivre, 2008).[2]

[2]**Spurious ambiguity** occurs when there are multiple derivations for the same dependency structure. This is the case in arc-standard dependency parsing: the structure $1 \leftarrow 2 \rightarrow 3$ can be obtained from two different action sequences (Cohen et al., 2012).

Typical features for transition-based dependency parsing include: the word and part-of-speech of the top element on the stack; the word and part-of-speech of the first, second, and third elements on the input buffer; pairs and triples of words and parts-of-speech from the top of the stack and the front of the buffer; the distance (in tokens) between the element on the top of the stack and the element in the front of the input buffer; the number of modifiers of each of these elements; and higher-order dependency features as described above in the section on graph-based dependency parsing. Zhang and Nivre (2011) describe a transition-based parser with rich features, which gave state-of-the-art performance (at the time) in both English and Chinese.

11.3.2 Pros and cons of transition-based dependency parsing

A key advantage of transition-based parsing is that it is much faster than graph-based methods. Since every word can be shifted once and every arc-creation action eliminates a word from the stack, the time complexity is linear in the length of the input. In contrast, graph-based parsing algorithms have quadratic or cubic time complexity.

Transition-based parsing can suffer from search errors, since an early mistake can make it impossible to find the best parse. This means that there could be an action sequence that would be preferred by the current parsing model, but is nonetheless not chosen because the first few actions in the sequence score badly. Put another way, transition-based parsing is **greedy** — unlike graph-based algorithms, which are guaranteed to find the best-scoring overall analysis. Solutions to this problem are discussed below.

Nonetheless, transition-based parsing achieves comparable accuracy to graph-based methods, in far less time (Nivre, 2004; Nivre et al., 2007). One reason is that in exchange for giving up on global inference (and thereby accepting the possibility of search errors), we free ourselves from any restrictions on the features that can be used in the classifier that selects each parsing action. For example, features may consider any number of previous parsing decisions, any aspect of the current stack, and any part of the input.

11.3.3 Alternative transition-based parsing algorithms

Arc-eager dependency parsing changes the ARC-RIGHT action so that right dependents can be attached before all of **their** dependents have been found. In arc-eager parsing, the ARC-RIGHT action creates an arc, and then pushes both the parent and child elements on to the stack. To remove these elements, it adds an addition REDUCE action, which can be applied to elements on the stack for whom an incoming edge has already been identified. 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).

Beam search A drawback of transition-based parsing is the possibility for search errors, in which a poor decision early in the parse will lead to cascading errors. **Beam search** is an improvement on greedy transition-based parsing, with the goal of eliminating search errors. As we move through the sentence, we keep a beam of possible hypotheses. For each element on the beam, we consider possible actions, and obtain a list of the top-k possiblities across all such actions. We then update the beam with the results of these top-k actions, and proceed (Zhang and Clark, 2008). Huang et al. (2012b) offer alternative perceptron learning rules that are specifically designed for learning in the beam search setting.

Shift-reduce parsing for CFGs Transition-based parsing can also be used for parsing in (binarized) context-free grammars. Here we use a shift-reduce parser, where each reduce operation creates a new non-terminal that produces the top two elements in the stack. When the input is consumed and the only element on the stack is a tree derived from the start symbol S, the input has been completely parsed.

11.3.4 Neural transition-based parsing

[todo: 2-3 paragraphs about the following papers:]

- Each shift-reduce decision is made by a locally-trained neural network (Chen and Manning, 2014). See also (Dyer et al., 2015)
- Shift-reduce decisions are made by neural network tranined on global conditional likelihood (Andor et al., 2016), using beam search.

11.4 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*[3]

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 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 11.8. This capability has implications for research in digital humanities, as shown by the analysis of Shakespeare performed by Muralidharan and Hearst (2013).

[3]Recall that the copula *is* is collapsed in many dependency grammars, such as the Universal Dependency treebank Nivre et al. (2016).

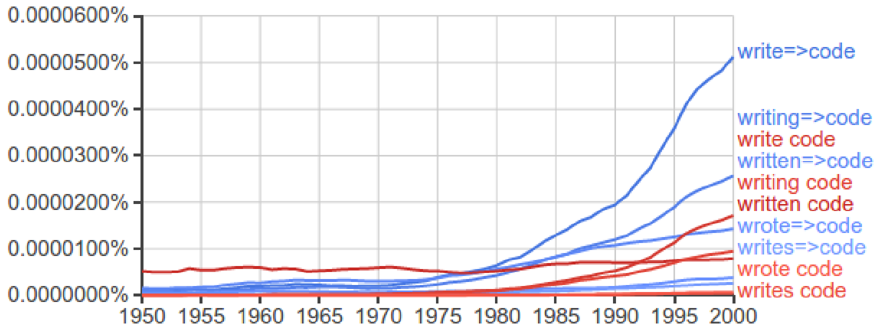


Figure 11.8: Google n-grams results for the bigram *write code* and the dependency arc *write => code* (and their morphological variants)

A classic application of dependency parsing is **relation extraction**, which is described in chapter 17. The goal of relation extraction is to identify entity pairs, such as

$$\begin{aligned}&\langle \text{Tolstoy, War and Peace}\rangle\\&\langle \text{Marquéz, 100 Years of Solitude}\rangle\\&\langle \text{Shakespeare, A Midsummer Night's Dream}\rangle,\end{aligned}$$

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,

(11.1) *What % of the nation's cheese does Wisconsin produce?*

Now suppose your corpus contains this sentence:

(11.2) *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.

(11.3) *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).

Exercises

1. The dependency structure $1 \leftarrow 2 \rightarrow 3$, with 2 as the root, can be obtained from more than one set of actions in arc-standard parsing. List both sets of actions that can obtain this parse.

Part III

Meaning

Chapter 12

Compositional semantics

A grand ambition of natural language processing is to convert natural language into a representation that supports inferences about meaning. Indeed, many potential applications of language technology involve some level of semantic understanding:

- Answering questions, such as *where is the nearest coffeeshop?* or *what is the middle name of the mother of the 44th President of the United States?*.
- Building a robot that can follow natural language instructions to execute tasks.
- Translating a sentence from one language into another, while preserving the underlying meaning.
- Fact-checking an article by searching the web for contradictory evidence.
- Logic-checking an argument by identifying contradictions, ambiguity, and unsupported assertions.

Each of these applications can be performed by converting natural language into a **meaning representation**. To be useful, a meaning representation must meet several criteria:

- **c1**: it should be unambiguous (unlike natural language);
- **c2**: it should provide a way to link language to external knowledge, observations, and actions;
- **c3**: it should support computational **inference**.

Much more than this can be said about the question of how best to represent knowledge for computation (e.g., Sowa, 2000), but we will focus on these three criteria.

12.1 Meaning and denotation

The first criterion for a meaning representation is that statements in the representation should be unambiguous — they should have only one possible interpretation. Natural language does not have this property: as we saw in chapter 9, sentences like *cats scratch people with claws* have multiple interpretations.

But what does it mean for a statement to be unambiguous? Programming languages provide a useful example: the output of a program is completely specified by the rules of the language, and the properties of the environment in which the program is run. For example, the python code `5 + 3` will have the output 8, as will the codes `(4*4)-(3*3)+1` and `((8))`. This output is known as the **denotation** of the program, and can be written as,

$$[\![\texttt{5+3}]\!] = [\![\texttt{(4*4)-(3*3)+1}]\!] = [\![\texttt{((8))}]\!] = 8. \tag{12.1}$$

In this sense, the program's output is its "meaning".

The denotations of these arithmetic expressions are determined by the meaning of the **constants** (e.g., `5`, `3`) and the **relations** (e.g., `+`, `*`, `(`, `)`). Now let's consider another snippet of python code, `double(4)`. The denotation of this code could be, $[\![\texttt{double(4)}]\!] = \texttt{8}$, or it could be $[\![\texttt{double(4)}]\!] = \texttt{44}$ — it depends on the meaning of `double`. This meaning is defined in a **world model** $\mathcal{M}$ as an infinite set of pairs. We write the denotation with respect to model $\mathcal{M}$ as $[\![\cdot]\!]_{\mathcal{M}}$, e.g., $[\![\texttt{double}]\!]_{\mathcal{M}} = \{\langle\texttt{0,0}\rangle, \langle\texttt{1,2}\rangle, \langle\texttt{2,4}\rangle, \ldots\}$. The world model would also define the (infinite) list of constants, e.g., $\{\texttt{0,1,2,...}\}$. As long as the denotation of string ϕ in model $\mathcal{M}$ can be computed unambiguously, the language can be said to be unambiguous.

This approach to meaning is known as **model-theoretic semantics**, and it addresses not only criterion $c1$ (no ambiguity), but also $c2$ (connecting language to external knowledge, observations, and actions). For example, we can connect a representation of the meaning of a statement like *the capital of Georgia* with a world model that includes knowledge base of geographical facts, obtaining the denotation `Atlanta`. We might populate a world model by applying an image analysis algorithm to Figure 12.1, and then use this world model to evaluate **propositions** like *a man is riding a moose*. Another desirable property of model-theoretic semantics is that when the facts change, the denotations change too: the meaning representation of *President of the USA* would have a different denotation in the model $\mathcal{M}_{2014}$ as it would in $\mathcal{M}_{2022}$.

12.2 Logical representations of meaning

If we can find a meaning representation which supports model-theoretic denotation, then we will have met criteria $c1$ and $c2$. The final criterion is $c3$, which requires that the meaning representation support inference — for example, automatically deducing new

Figure 12.1: A (doctored) image, which could be the basis for a world model [todo: the image is from 1912, so out of copyright? https://blogs.harvard.edu/houghton/2013/09/20/myths-debunked-sadly-theodore-roosevelt-never-rode-a-moose/]

facts from known premises. While many representations have been proposed that meet these criteria, the most mature is the language of first-order logic.[1]

12.2.1 Propositional logic

The bare bones of logical meaning representation are Boolean operations on propositions:

Propositional symbols Greek symbols like ϕ and ψ will be used to represent **propositions**, which are statements that are either true or false. For example, ϕ may correspond to the proposition, *bagels are delicious*.

Boolean operators We can build up more complex propositional formulas from Boolean operators. These include:

- Negation $\neg\phi$, which is true if ϕ is false.
- Conjunction, $\phi \wedge \psi$, which is true if both ϕ and ψ are true.

[1]Relevant alternatives include the "variable-free" representation used in semantic parsing of geographical queries (Zelle and Mooney, 1996) and robotic control (**?**), and dependency-based compositional semantics (Liang et al., 2013).

- Disjunction, $\phi \vee \psi$, which is true if at least one of P and Q is true
- Implication, $\phi \Rightarrow \psi$, which is true unless ϕ is true and ψ is false. Implication has identical truth conditions to $\neg\phi \vee \psi$.
- Equivalence, $\phi \Leftrightarrow \psi$, which is true if ϕ and ψ are both true or both false. Equivalence has identical truth conditions to $(\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \phi)$.

It is not strictly necessary to have all five Boolean operators: readers familiar with Boolean logic will know that it is possible to construct all other operators from either the NAND (not-and) or NOR (not-or) operators. Nonetheless, it is typical to use all five operators above for clarity. It is possible to define a number of "laws" for these Boolean operators, such as,

- **commutativity**: $\phi \wedge \psi = \psi \wedge \phi, \quad \phi \vee \psi = \psi \vee \phi$
- **associativity**: $\phi \wedge (\psi \wedge \chi) = (\phi \wedge \psi) \wedge \chi, \quad \phi \vee (\psi \vee \chi) = (\phi \vee \psi) \vee \chi$
- **complementation**: $\phi \wedge \neg\phi = \bot, \quad \phi \vee \neg\phi = \top$, where $\top$ indicates a true proposition and $\bot$ indicates a false proposition.

These laws can be combined to derive further equivalences, which can support logical inferences. For example, suppose $\phi =$ *The music is loud* and $\psi =$ *Max can't sleep*. Then if we are given,

$$\begin{array}{rl} \phi \Rightarrow \psi & \textit{If the music is loud, Max can't sleep.} \\ \phi & \textit{The music is loud.} \end{array}$$

we can derive ψ (*Max can't sleep*) by application of **modus ponens**, which is one of a set of **inference rules** that can be derived from more basic laws and used to manipulate propositional formulas. **Automated theorem provers** are capable of applying inference rules to a set of premises to derive desired propositions (Loveland, 2016).

12.2.2 First-order logic

Propositional logic is so named because it treats propositions as its base units. However, we would also like to reason about the content of the propositions themselves. For example,

(12.1) If anyone is making noise, then Max can't sleep.

(12.2) Abigail is making noise.

To understand the relationship between the statement *anyone is making noise* and the statement *Abigail is making noise*, we need some additional formal machinery. This is provided by **first-order logic** (FOL).

In FOL, logical propositions can be constructed from relationships between entities. Specifically, FOL extends propositional logic with the following classes of terms:

Constants These are elements that name individual entities in the model, such as MAX and ABIGAIL. The **denotation** of each constant in a model $\mathcal{M}$ is an element in the model, e.g., $[\![\text{MAX}]\!] = \texttt{m}$ and $[\![\text{ABIGAIL}]\!] = \texttt{a}$.

Relations Relations can be thought of as sets of entities, or sets of tuples. For example, the relation CAN-SLEEP is defined as the set of entities who can sleep, and has the denotation $[\![\text{CAN-SLEEP}]\!] = \{\texttt{a}, \texttt{m}, \ldots\}$. We can test the truth value of the proposition CAN-SLEEP(MAX) by asking whether $[\![\text{MAX}]\!] \in [\![\text{CAN-SLEEP}]\!]$. Logical relations that are defined over sets of entities are sometimes called **properties**.

Relations may also be ordered tuples of entities. For example BROTHER(MAX,ABIGAIL) expresses the proposition that MAX is the brother of ABIGAIL. The denotation of such relations is a set of tuples, $[\![\text{BROTHER}]\!] = \{\texttt{(m,a)}, \texttt{(x,y)}, \ldots\}$. We can test the truth value of the proposition BROTHER(MAX,ABIGAIL) by asking whether the tuple $([\![\text{MAX}]\!], [\![\text{ABIGAIL}]\!])$ is in the denotation $[\![\text{BROTHER}]\!]$.

We can now express statements like *Max can't sleep* and *Max is Abigail's brother*:

$$\neg\text{CAN-SLEEP}(\text{MAX})$$
$$\text{BROTHER}(\text{MAX},\text{ABIGAIL}).$$

We can also combine these statements using Boolean operators, such as,

$$(\text{BROTHER}(\text{MAX},\text{ABIGAIL}) \vee \text{BROTHER}(\text{MAX},\text{STEVE})) \Rightarrow \neg\text{CAN-SLEEP}(\text{MAX}).$$

We have thus far described a fragment of first-order logic, which permits only statements about specific entities. To support inferences about statements like *If **anyone** is making noise, then Max can't sleep*, we will need two additional elements in the meaning representation:

Variables These are mechanisms for referring to entities that are not locally specified. We can then write CAN-SLEEP(x) or BROTHER$(x$, ABIGAIL$)$. In these cases, x is an **free variable**, meaning that we have not committed to any particular assignment.

Quantifiers Variables are bound by quantifiers. There are two quantifiers in first-order logic.[2]

[2] In first-order logic, it is possible to quantify only over entities. In **second-order logic**, it is possible to quantify over properties, supporting statements like *Butch has every property that a good boxer has*,

$$\forall P \forall x((\text{GOOD-BOXER}(x) \Rightarrow P(x)) \Rightarrow P(\text{BUTCH}) \tag{12.2}$$

This example is from Blackburn and Bos (2005), who also show how first-order logic can "approximate" second-order and higher-order logics, by reifying sets as additional entities in the model.

- The **existential quantifier** $\exists$, which indicates that there must be at least one entity to which the variable can refer. For example, the statement $\exists x$MAKES-NOISE(X) indicates that there is at least one entity for which MAKES-NOISE is true.
- The **universal quantifier** $\forall$, which indicates that the the variable must be able to refer to any entity. For example, the statement,

$$\neg\text{MAKES-NOISE}(\text{ABIGAIL}) \Rightarrow (\forall x \text{CAN-SLEEP}(x)) \tag{12.3}$$

asserts that every entity can sleep if Abigail does not make noise.

The expressions $\exists x$ and $\forall x$ make x into a **bound variable**. A formula that contains no free variables is a **sentence**.

Functions Functions map from entities to entities, e.g., $[\![\text{CAPITAL-OF}(\text{GEORGIA})]\!] = [\![\text{ATLANTA}]\!]$. With functions, we also add an equality operator, so that it is possible to make statements like,

$$\forall x \exists y \text{MOTHER-OF}(x) = \text{DAUGHTER-OF}(y). \tag{12.4}$$

Note that MOTHER-OF is a functional analogue of the relation MOTHER, so that MOTHER-OF$(x) = y$ if MOTHER(x, y). Any logical formula that uses functions can be rewritten using only relations and quantification. For example,

$$\text{MAKES-NOISE}(\text{MOTHER-OF}(\text{ABIGAIL})) \tag{12.5}$$

can be rewritten as $\exists x$MAKES-NOISE$(x) \wedge$ MOTHER$(x$, ABIGAIL$)$.

An important property of quantifiers is that the order can matter. Unfortunately, natural language is rarely clear about this! The issue is demonstrated by examples like *everyone speaks a language*, which has the following interpretations:

$$\forall x \exists y \text{ SPEAKS}(x, y) \tag{12.6}$$
$$\exists y \forall x \text{ SPEAKS}(x, y). \tag{12.7}$$

In the first case, y may refer to several different languages, while in the second case, there is a single y that is spoken by everyone.

Truth-conditional semantics

One way to look at the meaning of an FOL sentence ϕ is as a set of **truth conditions**, or models under which ϕ is satisfied. But how can we determine whether a sentence in first-order logic is true or false? We will approach this inductively, starting with a predicate applied to a tuple of constants. The truth of such a sentence depends on whether the tuple of denotations of the constants is in the denotation of the predicate. For example, CAPITAL(GEORGIA,ATLANTA) is true in model $\mathcal{M}$ iff,

$$\langle [\![\text{GEORGIA}]\!]_{\mathcal{M}}, [\![\text{ATLANTA}]\!]_{\mathcal{M}} \rangle \in [\![\text{CAPITAL}]\!]_{\mathcal{M}}. \tag{12.8}$$

The Boolean operators $\wedge, \vee, \ldots$ provide ways to construct more complicated sentences, and the truth of such statements can be assessed based on the truth tables associated with these operators. The statement $\exists x \phi$ is true if there is some assignment of the variable x to an entity in the model such that ϕ is true; the statement $\forall x \phi$ is true if ϕ is true under all possible assignments of x. More formally, we would say that ϕ is **satisfied** under $\mathcal{M}$, written as $\mathcal{M} \models \phi$.

Truth conditional semantics allows us to define several other properties of sentences and pairs of sentences. Suppose that in every $\mathcal{M}$ under which ϕ is satisfied, another formula ψ is also satisfied; then ϕ **entails** ψ, sometimes written $\phi \models \psi$ [todo: double check]. For example,

$$\text{CAPITAL}(\text{GEORGIA},\text{ATLANTA}) \models \exists x \text{CAPITAL}(\text{GEORGIA}, x). \tag{12.9}$$

A statement that is satisfied under any model, such as $\phi \vee \neg\phi$, is **valid**; a statement that is not satisfied under any model, such as $\phi \wedge \neg\phi$, is **unsatisfiable**, or **inconsistent**. A **model checker** is a program that determines whether a sentence ϕ is satisfied in $\mathcal{M}$. A **model builder** is a program that constructs a model in which ϕ is satisfied. The problems of checking for consistency and validity in first-order logic are **undecidable**, meaning that there is no algorithm that can automatically determine whether an FOL formula is valid or inconsistent.

Inference in first-order logic

Our original goal was to support inferences that combine general statements *If anyone is making noise, then Max can't sleep* with specific statements like *Abigail is making noise*. We can now represent such statements in first-order logic, but how are we to perform the inference that *Max can't sleep*? One approach is to use "generalized" versions of propositional inference rules like modus ponens, which can be applied to FOL formulas. By repeatedly applying such inference rules to a knowledge base of facts, it is possible to produce proofs of desired propositions. To find the right sequence of inferences to derive a desired theorem, classical artificial intelligence search algorithms like backward chaining can be applied. Such algorithms are implemented in interpreters for the `prolog` logic programming language (Pereira and Shieber, 2002).

12.3 Semantic parsing and the lambda calculus

The previous section has laid out a lot of formal machinery, which we will now try to unite with natural language. Given an English sentence like *Alex likes Brit*, how can we

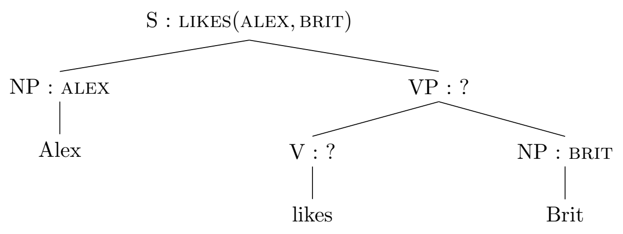


Figure 12.2: The principle of compositionality requires that we identify meanings for the constituents *likes* and *likes Brit* that will make it possible to compute the meaning for the entire sentence.

obtain the desired first-order logical representation, LIKES(ALEX,BRIT)? This is the task of **semantic parsing**. Just as a syntactic parser is a function from a natural language sentence to a syntactic structure such as a phrase structure tree, a semantic parser is a function from natural language to logical formulas.

As in syntactic analysis, semantic parsing is difficult because the space of inputs and outputs is very large, and their interaction is complex. Our best hope is that, like syntactic parsing, semantic parsing can somehow be decomposed into simpler sub-problems. This idea, usually attributed to the German philosopher Gottlob Frege, is called the **principle of compositionality**: the meaning of a complex expression is a function of the meanings of that expression's constituent parts. We will define these "constituent parts" as syntactic elements like noun phrases and verb phrases. These constituents are combined using function application: if the syntactic parse contains the production $x \rightarrow y\ z$, then the semantics of x, written x.sem, will be computed as a function of the semantics of the constituents, y.sem and z.sem.[3] [4]

12.3.1 The lambda calculus

Let's see how this works for a simple sentence like *Alex likes Brit*, whose syntactic structure is shown in Figure 12.2. Our goal is the formula, LIKES(ALEX,BRIT), and it is clear that the meaning of the constituents *Alex* and *Brit* should be ALEX and BRIT. That leaves two more constituents: the verb *likes*, and the verb phrase *likes Brit*. How can we define the meaning of these units in a way that enables us to recover the desired meaning for the entire sentence? If the meanings of *Alex* and *Brit* are constants, then the meanings of *likes*

[3]chapter 9 briefly discusses alternative syntactic formalisms, including Combinatory Categorial Grammar (CCG). CCG is argued to be particularly well-suited to semantic parsing (Hockenmaier and Steedman, 2007), and is used in much of the contemporary work on machine learning for semantic parsing, summarized in § 12.4.

[4]The approach of algorithmically building up meaning representations from a series of operations on the syntactic structure of a sentence is generally attributed to the philosopher Richard Montague, who published a series of influential papers on the topic in the early 1970s (e.g., Montague, 1973).

and *likes Brit* must be functional expressions, which can be applied to their siblings to produce the desired analyses.

Modeling these partial analyses requires extending the first-order logic meaning representation. We do this by adding **lambda expressions**, which are descriptions of anonymous functions,[5] e.g.,

$$\lambda x.\text{LIKES}(x, \text{BRIT}). \tag{12.10}$$

We take this functional expression to be the meaning of the verb phrase *likes Brit*; it takes a single argument, and returns the result of substituting that argument for x in the expression $\text{LIKES}(x, \text{BRIT})$. We write this substitution as,

$$(\lambda x.\text{LIKES}(x, \text{BRIT}))@\text{ALEX} = \text{LIKES}(\text{ALEX},\text{BRIT}), \tag{12.11}$$

with the symbol "@" indicating function application. Function application in the lambda calculus is sometimes called **β-reduction** or **β-conversion**. We will write $\phi@\psi$ to indicate a function application to be performed by β-reduction, and $\phi(\psi)$ to indicate a function or predicate in the final logical form.

Equation 12.11 shows how to obtain the desired semantics for the sentence *Alex likes Brit*: by applying the lambda expression $\lambda x.\text{LIKES}(x, \text{BRIT})$ to the logical constant ALEX. This rule of composition can be specified in a syntactic-semantic grammar: for the syntactic production S $\rightarrow$ NP VP, we have the semantic rule VP.sem@NP.sem.

The meaning of the meaning of the transitive verb phrase can also be obtained by function application on its syntactic constituents. For the syntactic production VP $\rightarrow$ V NP, we apply the semantic rule,

$$\text{VP.sem} = (\text{V.sem})@\text{NP.sem} \tag{12.12}$$
$$= (\lambda y.\lambda x.\text{LIKES}(x, y))@(\text{BRIT}) \tag{12.13}$$
$$= \lambda x.\text{LIKES}(x, \text{BRIT}). \tag{12.14}$$

Here we have defined the meaning of the transitive verb *likes* as a lambda expression whose output is **another** lambda expression: it takes y as an argument to fill in one of the slots in the LIKES relation, and returns a lambda expression that is ready to take an argument to fill in the other slot.[6]

Table 12.1 shows a minimal syntactic/semantic grammar fragment, which we will call G_1. The complete **derivation** of *Alex likes Brit* in G_1 is shown in Figure 12.3. In addition to

[5]Formally, all first-order logic formulas are lambda expressions; in addition, if ϕ is a lambda expression, then $\lambda x.\phi$ is also a lambda expression. Readers who are familiar with functional programming will recognize lambda expressions from their use in programming languages such as Lisp and Python.

[6]This can be written in a few different ways. More informally, we can write $\lambda y, x.\text{LIKES}(x, y)$, indicating a lambda expression that takes two arguments; this would be acceptable in functional programming. More formally, logicians (e.g., Carpenter, 1997) often write $\lambda y.\lambda x.\text{LIKES}(x)(y)$, indicating that each lambda expression takes exactly one argument.

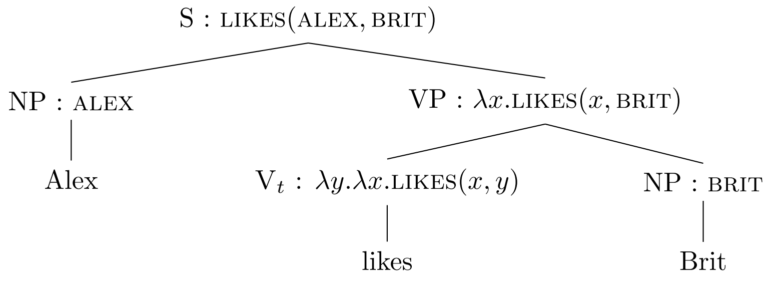

S : LIKES(ALEX, BRIT)
NP : ALEX
VP : λx.LIKES(x, BRIT)
Alex
V_t : $\lambda y.\lambda x$.LIKES(x, y)
NP : BRIT
likes
Brit

Figure 12.3: Derivation of the semantic representation for *Alex likes Brit* in the grammar G_1.

S	$\rightarrow$ NP VP	VP.sem@NP.sem
VP	$\rightarrow$ V_t NP	V_t.sem@NP.sem
VP	$\rightarrow$ V_i	V_i.sem
V_t	$\rightarrow$ *likes*	$\lambda y.\lambda x.\text{LIKES}(x, y)$
V_i	$\rightarrow$ *sleeps*	$\lambda x.\text{SLEEPS}(x)$
NP	$\rightarrow$ *Alex*	ALEX
NP	$\rightarrow$ *Brit*	BRIT

Table 12.1: G_1, a minimal syntactic/semantic context-free grammar

the transitive verb *likes*, the grammar also includes the intransitive verb *sleeps*; it should be clear how to derive the meaning of sentences like *Alex sleeps*. For verbs that can be either transitive or intransitive, such as *eats*, we would have two terminal productions, one for each sense (terminal productions are also called the **lexical entries**). Indeed, most of the grammar is in the **lexicon** (the terminal productions), since these productions select the basic units of the semantic interpretation.

12.3.2 Quantification

Things get more complicated when we move from sentences about named entities to sentences that involve more general noun phrases. Let's consider the example, *A dog sleeps*, which has the meaning $\exists x \text{DOG}(x) \wedge \text{SLEEPS}(x)$. Clearly, the DOG relation will be introduced by the word *dog*, and the SLEEP relation will be introduced by the word *sleeps*. The existential quantifier $\exists$ must be introduced by the lexical entry for the determiner *a*.[7]

[7]Conversely, the sentence *Every dog sleeps* would involve a universal quantifier, $\forall x \text{DOG}(x) \Rightarrow \text{SLEEPS}(x)$. The definite article *the* requires more consideration, since *the dog* must refer to some dog which is uniquely identifiable, perhaps from contextual information external to the sentence. Carpenter (1997, pp. 96-100) summarizes recent approaches to handling definite descriptions.

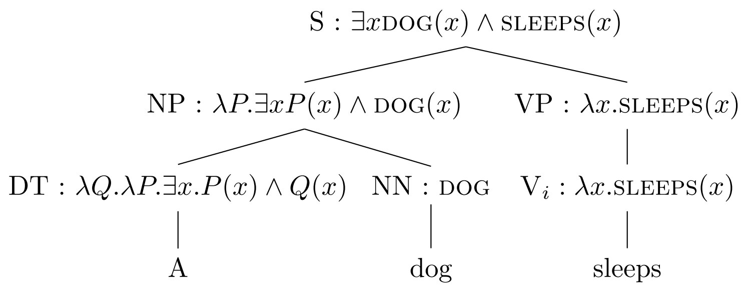

S : $\exists x \textsc{dog}(x) \wedge \textsc{sleeps}(x)$
NP : $\lambda P.\exists x P(x) \wedge \textsc{dog}(x)$
VP : $\lambda x.\textsc{sleeps}(x)$
DT : $\lambda Q.\lambda P.\exists x.P(x) \wedge Q(x)$
NN : $\textsc{dog}$
V$_i$: $\lambda x.\textsc{sleeps}(x)$
A
dog
sleeps

Figure 12.4: Derivation of the semantic representation for *A dog sleeps*, in grammar G_2

However, this seems problematic for the compositional approach taken in the grammar G_1: if the semantics of the noun phrase *a dog* is an existentially quantified expression, how can it be the argument to the semantics of the verb *sleeps*, which expects an entity? And where does the logical conjunction come from?

There are a few different approaches to handling these issues.[8] We will begin by reversing the semantic relationship between subject NPs and VPs, so that the production S $\rightarrow$ NP VP has the semantics NP.sem@VP.sem: the meaning of the sentence is now the semantics of the noun phrase applied to the verb phrase. The implications of this change are best illustrated by exploring the derivation of the example, shown in Figure 12.4. Let's start with the indefinite article *a*, to which we assign the rather intimidating semantics,

$$\lambda P.\lambda Q.\exists x P(x) \wedge Q(x). \tag{12.15}$$

This is a lambda expression that takes two **relations** as arguments, P and Q. The relation P is scoped to the outer lambda expression, so it will be provided by the immediately adjacent noun, which in this case is $\textsc{dog}$. Thus, the noun phrase *a dog* has the semantics,

$$\begin{aligned}
\text{NP.sem} =& \textsc{Det}.\text{sem}@\textsc{Nn}.\text{sem} && (12.16)\\
=& (\lambda P.\lambda Q.\exists x P(x) \wedge Q(x))@(\textsc{dog}) && (12.17)\\
=& \lambda Q.\exists x \textsc{dog}(x) \wedge Q(x). && (12.18)
\end{aligned}$$

This is a lambda expression that is expecting another relation, Q, which will be provided by the verb phrase, $\textsc{sleeps}$. This gives the desired analysis, $\exists x \textsc{dog}(x) \wedge \textsc{sleeps}(x)$.[9]

If noun phrases like *a dog* are interpreted as lambda expressions, then proper nouns like *Alex* must be treated in the same way. This is achieved by **type-raising** from constants to lambda expressions, $x \Rightarrow \lambda P.P(x)$. After type-raising, the semantics of *Alex* is

[8]Carpenter (1997) offers an alternative treatment based on combinatory categorial grammar.

[9]When applying β-reduction to arguments that are themselves lambda expressions, be sure to use unique variable names to avoid confusion. For example, it is important to distinguish the x in the semantics for *a* from the x in the semantics for *likes*. Variable names are abstractions, and can always be changed — this is known as α**-conversion**. For example, $\lambda x.P(x)$ can be converted to $\lambda y.P(y)$, etc.

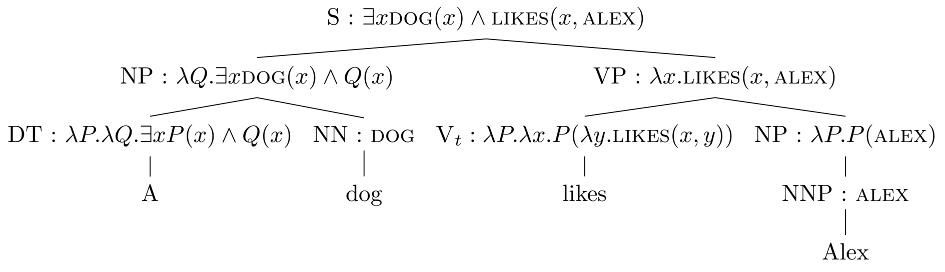


Figure 12.5: Derivation of the semantic representation for *A dog likes Alex*.

$\lambda P.P(\text{ALEX})$ — a lambda expression that expects a relation to tell us something about ALEX.[10] Make sure you see how the analysis in Figure 12.4 can be applied to the sentence *Alex sleeps*.

To handle direct objects, we will perform the same type-raising operation on transitive verbs: the meaning of verbs such as *likes* will be raised to,

$$\lambda P.\lambda x.P(\lambda y.\text{LIKES}(x, y)) \tag{12.19}$$

As a result, we can keep the verb phrase production VP.sem = V.sem@NP.sem, knowing that the direct object will provide the function P in (12.19). To see how this works, let's analyze the verb phrase *likes a dog*. After uniquely relabeling each lambda variable, we have,

$$\begin{aligned}
\text{VP.sem} =& \text{V.sem@NP.sem} \\
=& (\lambda P.\lambda x.P(\lambda y.\text{LIKES}(x, y)))@(\lambda Q.\exists z\text{DOG}(z) \wedge Q(z)) \\
=& \lambda x.(\lambda Q.\exists z\text{DOG}(z) \wedge Q(z))@(\lambda y.\text{LIKES}(x, y)) \\
=& \lambda x.\exists z\text{DOG}(z) \wedge (\lambda y.\text{LIKES}(x, y))@z \\
=& \lambda x.\exists z\text{DOG}(z) \wedge \text{LIKES}(x, z).
\end{aligned}$$

These changes are summarized in the revised grammar G_2, shown in Table 12.2. Figure 12.5 shows a derivation that involves a transitive verb, an indefinite noun phrase, and a proper noun.

[10]Compositional semantic analysis is often supported by **type systems**, which make it possible to check whether a given function application is valid. The base types are entities e and truth values t. A property, such as DOG, is a function from entities to truth values, so its type is written $\langle e, t\rangle$. A transitive verb has type $\langle e, \langle e, t\rangle\rangle$: after receiving the first entity (the direct object), it returns a function from entities to truth values, which will be applied to the subject of the sentence. The type-raising operation $x \Rightarrow \lambda P.P(x)$ corresponds to a change in type from e to $\langle\langle e, t\rangle, t\rangle$: it expects a function from entities to truth values, and returns a truth value.

S	$\rightarrow$ NP VP	NP.sem@VP.sem
VP	$\rightarrow$ V_t NP	V_t.sem@NP.sem
VP	$\rightarrow$ V_i	V_i.sem
NP	$\rightarrow$ DET NN	DET.sem@NN.sem
NP	$\rightarrow$ NNP	$\lambda P.P(\text{NNP.sem})$
DET	$\rightarrow$ *a*	$\lambda P.\lambda Q.\exists x P(x) \wedge Q(x)$
DET	$\rightarrow$ *every*	$\lambda P.\lambda Q.\forall x(P(x) \Rightarrow Q(x))$
V_t	$\rightarrow$ *likes*	$\lambda P.\lambda x.P(\lambda y.\text{LIKES}(x, y))$
V_i	$\rightarrow$ *sleeps*	$\lambda x.\text{SLEEPS}(x)$
NN	$\rightarrow$ *dog*	DOG
NNP	$\rightarrow$ *Alex*	ALEX
NNP	$\rightarrow$ *Brit*	BRIT

Table 12.2: G_2, a syntactic/semantic context-free grammar fragment, which supports quantified noun phrases

12.4 Learning semantic parsers

As with syntactic parsing, any syntactic/semantic grammar with sufficient coverage will **overgenerate**, producing many possible analyses for any given sentence. Machine learning is the dominant approach to selecting a single analysis. We will focus on algorithms that learn to score logical forms by attaching weights to features of their derivations (Zettlemoyer and Collins, 2005). Alternative approaches include transition-based parsing (Zelle and Mooney, 1996; Misra and Artzi, 2016) and methods inspired by machine translation (Wong and Mooney, 2006). Methods also differ in the form of supervision used for learning, which can range from complete derivations to much more limited training signals. We will begin with the case of complete supervision, and then consider how learning is still possible even when seemingly key information is missing.

Datasets Early work on semantic parsing focused on geographical queries, such as *What states border Texas*. The GeoQuery dataset of Zelle and Mooney (1996) was originally coded in prolog, but has subsequently been expanded and converted into the SQL database query language by Popescu et al. (2003) and into first-order logic with lambda calculus by Zettlemoyer and Collins (2005), providing logical forms like $\lambda x.\text{STATE}(x) \wedge \text{BORDERS}(x, \text{TEXAS})$. Another early dataset consists of instructions for RoboCup robot soccer teams (Kate et al., 2005). More recent work has focused on broader domains, such as the Freebase database (Bollacker et al., 2008), for which queries have been annotated by Krishnamurthy and Mitchell (2012) and Cai and Yates (2013), as well on child-directed speech (Kwiatkowski et al., 2012) and elementary school science exams (Krishnamurthy, 2016).

12.4.1 Learning from derivations

Let $\boldsymbol{w}^{(i)}$ indicate a sequence of text, and let $\boldsymbol{y}^{(i)}$ indicate the desired logical form. For example:

$$\begin{aligned}\boldsymbol{w}^{(i)} =& \text{Alex eats shoots and leaves}\\ \boldsymbol{y}^{(i)} =& \text{EATS(ALEX,SHOOTS)} \wedge \text{EATS(ALEX,LEAVES)}\end{aligned}$$

In the standard supervised learning paradigm that was introduced in chapter 2, we first define a feature function, $\boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y})$, and then learn weights on these features, so that $\boldsymbol{y}^{(i)} = \operatorname{argmax}_{\boldsymbol{y}} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y})$. The weight vector $\boldsymbol{\theta}$ is learned by comparing the features of the true label $\boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)})$ against either the features of the predicted label $\boldsymbol{f}(\boldsymbol{w}^{(i)}, \hat{\boldsymbol{y}})$ (perceptron, support vector machine) or the expected feature vector $E_{\boldsymbol{y}|\boldsymbol{w}}[\boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y})]$ (logistic regression).

While this basic framework seems similar to discriminative syntactic parsing, there is a crucial difference. In (context-free) syntactic parsing, the annotation $\boldsymbol{y}^{(i)}$ contains all of the syntactic productions; indeed, the task of identifying the correct set of productions is identical to the task of identifying the syntactic structure. In semantic parsing, this is not the case: the logical form EATS(ALEX,SHOOTS) $\wedge$ EATS(ALEX,LEAVES) does not reveal the syntactic/semantic productions that were used to obtain it. Indeed, there may be **spurious ambiguity**, so that a single logical form can be reached by multiple derivations. (We previously encountered spurious ambiguity in transition-based dependency parsing, § 11.3.1.)

Let us introduce an additional variable $\boldsymbol{z}$, representing the **derivation** of the logical form $\boldsymbol{y}$ from the text $\boldsymbol{w}$. We assume that the feature function decomposes across the productions in the derivation, $\boldsymbol{f}(\boldsymbol{w}, \boldsymbol{z}, \boldsymbol{y}) = \sum_{t=1}^{T} \boldsymbol{f}(\boldsymbol{w}, z_t, \boldsymbol{y})$, where z_t indicates a single syntactic/semantic production. For example, we might have a feature for the high-level production S $\rightarrow$ NP VP : NP.sem@VP.sem, as well as for terminal productions like NNP $\rightarrow$ *Alex* : ALEX. Under this decomposition, we can compute scores for each semantically-annotated subtree in the analysis of $\boldsymbol{w}$, and can therefore apply bottom-up parsing algorithms like CKY (chapter 10) to find the best-scoring semantic analysis.

Figure 12.6 shows a derivation of the correct semantic analysis of the sentence *Alex eats shoots and leaves*, in a simplified grammar in which the plural noun phrases *shoots* and *leaves* are interpreted as logical constants SHOOTS and LEAVES$_n$. Figure 12.7 shows a derivation of an incorrect analysis. Assuming one feature per production, the perceptron update is shown in Table 12.3. From this update, the parser would learn to prefer the noun interpretation of *leaves* over the verb interpretation. It would also learn to prefer noun phrase coordination over verb phrase coordination. Note that we could easily replace the perceptron with a conditional random field. In this case, the online updates would be based on feature expectations, which can be computed using bottom-up algorithms like inside-outside (§ 10.5.2).

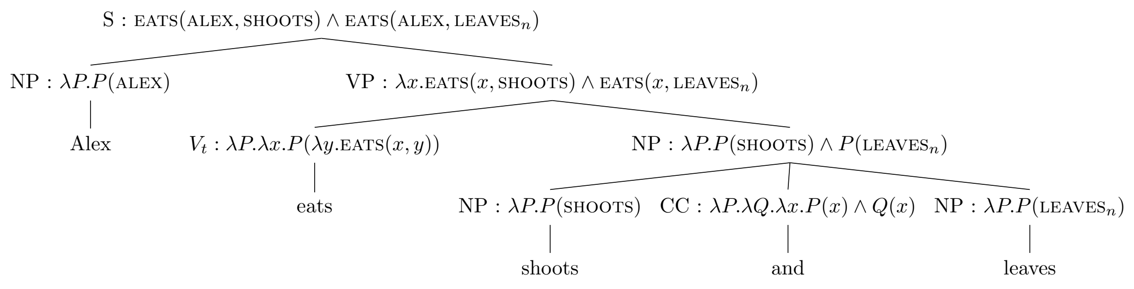

S : EATS(ALEX, SHOOTS) ∧ EATS(ALEX, LEAVES$_n$)
NP : $\lambda P.P$(ALEX)
Alex
VP : $\lambda x.$EATS(x, SHOOTS) ∧ EATS(x, LEAVES$_n$)
V$_t$: $\lambda P.\lambda x.P(\lambda y.$EATS$(x,y))$
eats
NP : $\lambda P.P$(SHOOTS) ∧ P(LEAVES$_n$)
NP : $\lambda P.P$(SHOOTS)
shoots
CC : $\lambda P.\lambda Q.\lambda x.P(x) \wedge Q(x)$
and
NP : $\lambda P.P$(LEAVES$_n$)
leaves

Figure 12.6: Derivation for gold semantic analysis of *Alex eats shoots and leaves*

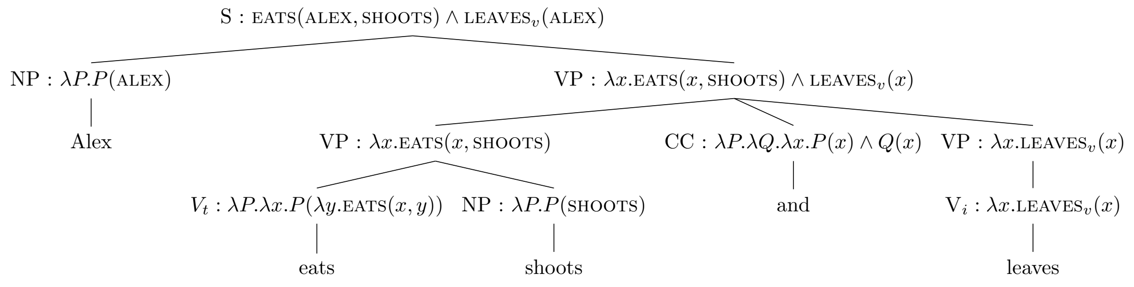

S : EATS(ALEX, SHOOTS) ∧ LEAVES$_v$(ALEX)
NP : $\lambda P.P$(ALEX)
Alex
VP : $\lambda x.$EATS(x, SHOOTS) ∧ LEAVES$_v$(x)
VP : $\lambda x.$EATS(x, SHOOTS)
V$_t$: $\lambda P.\lambda x.P(\lambda y.$EATS$(x,y))$
eats
NP : $\lambda P.P$(SHOOTS)
shoots
CC : $\lambda P.\lambda Q.\lambda x.P(x) \wedge Q(x)$
and
VP : $\lambda x.$LEAVES$_v$(x)
V$_i$: $\lambda x.$LEAVES$_v$(x)
leaves

Figure 12.7: Derivation for incorrect semantic analysis of *Alex eats shoots and leaves*

$\text{NP}_1 \rightarrow \text{NP}_2\ \text{Cc}\ \text{NP}_3$	(Cc.sem@(NP$_2$.sem))@(NP$_3$.sem)	+1
$\text{VP}_1 \rightarrow \text{VP}_2\ \text{Cc}\ \text{VP}_3$	(Cc.sem@(VP$_2$.sem))@(VP$_3$.sem)	-1
NP → *leaves*	LEAVES$_n$	+1
VP → V$_i$	V$_i$.sem	-1
V$_i$ → *leaves*	$\lambda x.$LEAVES$_v$	-1

Table 12.3: Perceptron update for analysis in Figure 12.6 (gold) and Figure 12.7 (predicted)

12.4.2 Learning from logical forms

Complete derivations are expensive to annotate, and are rarely available.[11] More recent work has focused on learning from logical forms directly, while treating the derivations as **latent variables** (Zettlemoyer and Collins, 2005). In a conditional probabilistic model over logical forms $\boldsymbol{y}$ and derivations $\boldsymbol{z}$, we have,

$$\text{p}(\boldsymbol{y}, \boldsymbol{z} \mid \boldsymbol{w}) = \frac{\exp(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{z}, \boldsymbol{y}))}{\sum_{\boldsymbol{y}', \boldsymbol{z}'} \exp(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{z}', \boldsymbol{y}'))}, \tag{12.20}$$

[11] An exception is the work of Ge and Mooney (2005), who annotate the meaning of each syntactic constituents for several hundred sentences.

which is the standard log-linear model, applied to the logical form $\boldsymbol{y}$ and the derivation $\boldsymbol{z}$.

Since the derivation $\boldsymbol{z}$ completely determines the logical form $\boldsymbol{y}$, it may seem silly to model the joint probability over $\boldsymbol{y}$ and $\boldsymbol{z}$. However, since $\boldsymbol{z}$ is unknown, it can be marginalized out,

$$\mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w}) = \sum_{\boldsymbol{z}} \mathrm{p}(\boldsymbol{y}, \boldsymbol{z} \mid \boldsymbol{w}). \tag{12.21}$$

We can then have the semantic parser select the logical form with the maximum log marginal probability,

$$\begin{aligned}
\log \sum_{\boldsymbol{z}} \mathrm{p}(\boldsymbol{y}, \boldsymbol{z} \mid \boldsymbol{w}) =& \log \sum_{\boldsymbol{z}} \frac{\exp(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{z}, \boldsymbol{y}))}{\sum \boldsymbol{y}', \boldsymbol{z}' \exp(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{z}', \boldsymbol{y}'))} && (12.22)\\
&\propto \log \sum_{\boldsymbol{z}} \exp(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{z}', \boldsymbol{y}')) && (12.23)\\
&\geq \max_{\boldsymbol{z}} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{z}, \boldsymbol{y}). && (12.24)
\end{aligned}$$

Note that it is impossible to push the $\log$ term inside the sum over $\boldsymbol{z}$, meaning that our usual linear scoring function does not apply. We can recover this scoring function only in approximation, by taking the max (rather than the sum) over derivations $\boldsymbol{z}$, which provides a lower bound.

Learning can be performed by maximizing the log marginal likelihood,

$$\begin{aligned}
\ell(\boldsymbol{\theta}) =& \sum_{i=1}^{N} \log \mathrm{p}(\boldsymbol{y}^{(i)} \mid \boldsymbol{w}^{(i)}; \boldsymbol{\theta}) && (12.25)\\
=& \sum_{i=1}^{N} \log \sum_{\boldsymbol{z}} \mathrm{p}(\boldsymbol{y}^{(i)}, \boldsymbol{z}^{(i)} \mid \boldsymbol{w}^{(i)}; \boldsymbol{\theta}). && (12.26)
\end{aligned}$$

This log-likelihood is not **convex** in $\boldsymbol{\theta}$, unlike the log-likelihood of a fully-observed conditional random field. This means that learning can give different results depending on the initialization.

The derivative of (12.26) is,

$$\begin{aligned}
\frac{\partial \ell_i}{\partial \boldsymbol{\theta}} =& \sum_{\boldsymbol{z}} \mathrm{p}(\boldsymbol{z} \mid \boldsymbol{y}, \boldsymbol{w}; \boldsymbol{\theta}) \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{z}, \boldsymbol{y}) - \sum_{\boldsymbol{y}', \boldsymbol{z}'} \mathrm{p}(\boldsymbol{y}', \boldsymbol{z}' \mid \boldsymbol{w}; \boldsymbol{\theta}) \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{z}', \boldsymbol{y}') && (12.27)\\
=& E_{\boldsymbol{z} \mid \boldsymbol{y}, \boldsymbol{w}} \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{z}, \boldsymbol{y}) - E_{\boldsymbol{y}, \boldsymbol{z} \mid \boldsymbol{w}} \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{z}, \boldsymbol{y}) && (12.28)
\end{aligned}$$

Both expectations can be computed via bottom-up algorithms like inside-outside. Alternatively, we can again maximize rather than marginalize over derivations for an approximate solution. In either case, the first term of the gradient requires us to identify

Algorithm 11 Latent variable perceptron

```
1: procedure LatentVariablePerceptron($\boldsymbol{w}^{(1:N)}, \boldsymbol{y}^{(1:N)}$)
2:     $\boldsymbol{\theta} \leftarrow \mathbf{0}$
3:     repeat
4:         Select an instance $i$
5:         $\boldsymbol{z}^{(i)} \leftarrow \text{argmax}_{\boldsymbol{z}}\, \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{z}, \boldsymbol{y}^{(i)})$
6:         $\hat{\boldsymbol{y}}, \hat{\boldsymbol{z}} \leftarrow \text{argmax}_{\boldsymbol{y}',\boldsymbol{z}'}\, \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{z}', \boldsymbol{y}')$
7:         $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{z}^{(i)}, \boldsymbol{y}^{(i)}) - \boldsymbol{f}(\boldsymbol{w}^{(i)}, \hat{\boldsymbol{z}}, \hat{\boldsymbol{y}})$
8:     until tired
9:     return $\boldsymbol{\theta}$
```

derivations $\boldsymbol{z}$ that are compatible with the logical form $\boldsymbol{y}$. This can be done in a bottom-up dynamic programming algorithm, by having each cell in the table $t[i, j]$ include both the constituent types (e.g., NP, S) that can derive the span $\boldsymbol{w}_{i:j-1}$, as well as the set of possible logical forms. The inclusion of logical forms means that the resulting table may be much larger than in syntactic parsing, where it is limited by the number of non-terminals in the grammar. This can be controlled by using pruning to eliminate intermediate analyses that are incompatible with the final logical form $\boldsymbol{y}$ (Zettlemoyer and Collins, 2005), or by using beam search and restricting the size of each cell to some fixed constant (Liang et al., 2013).

If we replace each expectation in (12.28) with argmax and then apply stochastic gradient descent to learn the weights, we obtain the **latent variable perceptron**, a simple and general algorithm for learning with missing data. The algorithm is shown in its most basic form in Algorithm 11, but the usual tricks such as averaging and margin loss can be applied (Yu and Joachims, 2009). Aside from semantic parsing, the latent variable perceptron has been used in tasks such as machine translation (Liang et al., 2006) and named entity recognition (Sun et al., 2009). In **latent conditional random fields**, we use the full expectations rather than maximizing over the hidden variable. This model has also been employed in a range of problems beyond semantic parsing, including parse reranking (Koo and Collins, 2005) and gesture recognition (Quattoni et al., 2007).

12.4.3 Learning from denotations

Logical forms are easier to obtain than complete derivations, but the annotation of logical forms still requires considerable expertise. However, it is relatively easy to obtain denotations for many natural language sentences. For example, in the geography domain, the denotation of a question would be its answer (Clarke et al., 2010; Liang et al., 2013):

$$\begin{aligned}\textbf{Text} :& \textit{What states border Georgia?}\\ \textbf{Logical form} :& \lambda x.\text{STATE}(x) \wedge \text{BORDER}(x, \text{GEORGIA})\\ \textbf{Denotation} :& \{\texttt{Alabama, Florida, North Carolina,}\\ & \texttt{South Carolina, Tennessee}\}\end{aligned}$$

Similarly, in a robotic control setting, the denotation of a command would be an action or sequence of actions (Artzi and Zettlemoyer, 2013). In both cases, the idea is to reward the semantic parser for choosing an analysis whose denotation is correct: the right answer to the question, or the right action.

Learning from logical forms was made possible by summing or maxing over derivations. This idea can be carried one step further, summing or maxing over all logical forms with the correct denotation. Let $v_i(\boldsymbol{y}) \in \{0, 1\}$ be a **validation function**, which assigns a binary score indicating whether the denotation $[\![\boldsymbol{y}]\!]$ for the text $\boldsymbol{w}^{(i)}$ is correct. We can then learn by maximizing a conditional-likelihood objective,

$$\ell^{(i)}(\boldsymbol{\theta}) = \log \sum_{\boldsymbol{y}} v_i(\boldsymbol{y}) \times \text{p}(\boldsymbol{y} \mid \boldsymbol{w}; \boldsymbol{\theta}) \tag{12.29}$$

$$= \log \sum_{\boldsymbol{y}} v_i(\boldsymbol{y}) \times \sum_{\boldsymbol{z}} \text{p}(\boldsymbol{y}, \boldsymbol{z} \mid \boldsymbol{w}; \boldsymbol{\theta}), \tag{12.30}$$

which sums over all derivations $\boldsymbol{z}$ of all valid logical forms, $\{\boldsymbol{y} : v_i(\boldsymbol{y}) = 1\}$. This corresponds to the log-probability that the semantic parser produces a logical form with a valid denotation.

Differentiating with respect to $\boldsymbol{\theta}$, we obtain,

$$\frac{\partial \ell^{(i)}}{\partial \boldsymbol{\theta}} = \sum_{\boldsymbol{y},\boldsymbol{z}:v_i(\boldsymbol{y})=1} \text{p}(\boldsymbol{y}, \boldsymbol{z} \mid \boldsymbol{w})\boldsymbol{f}(\boldsymbol{w}, \boldsymbol{z}, \boldsymbol{y}) - \sum_{\boldsymbol{y}',\boldsymbol{z}'} \text{p}(\boldsymbol{y}', \boldsymbol{z}' \mid \boldsymbol{w})\boldsymbol{f}(\boldsymbol{w}, \boldsymbol{z}', \boldsymbol{y}'), \tag{12.31}$$

which is the usual difference in feature expectations. The positive term computes the expected feature expectations conditioned on the denotation being valid, while the second term computes the expected feature expectations according to the current model, without regard to the ground truth. Large-margin learning formulations are also possible for this problem. For example, Artzi and Zettlemoyer (2013) generate a set of valid and invalid derivations, and then impose a constraint that all valid derivations should score higher than all invalid derivations. This constraint drives a perceptron-like learning rule.

Additional resources

A key issue not considered here is how to handle **semantic underspecification**: cases in which there are multiple semantic interpretations for a single syntactic structure. Quantifier scope ambiguity is a classic example. Blackburn and Bos (2005) enumerate a number of approaches to this issue, and also provide links between natural language semantics and computational inference techniques. Much of the contemporary research on semantic parsing uses the framework of combinatory categorial grammar (CCG). Carpenter (1997) provides a comprehensive treatment of how CCG can support compositional semantic analysis. Another recent area of research is the semantics of multi-sentence texts. This can be handled with models of **dynamic semantics**, such as dynamic predicate logic (Groenendijk and Stokhof, 1991).

To learn more about ongoing research on data-driven semantic parsing, readers may consult the survey article by Liang and Potts (2015), tutorial slides and videos by Artzi and Zettlemoyer (2013),[12] and the source code by Yoav Artzi[13] and Percy Liang.[14]

Exercises

1. Derive the **modus ponens** inference rule, which states that if we know $\phi \Rightarrow \psi$ and ϕ, then ψ must be true. The derivation can be performed using the definition of the $\Rightarrow$ operator and some of the laws provided in § 12.2.1, plus one additional identity: $\bot \vee \phi = \phi$.

2. Convert the following examples into first-order logic, using the relations CAN-SLEEP, MAKES-NOISE, and BROTHER.

   - If Abigail makes noise, no one can sleep.
   - If Abigail makes noise, someone cannot sleep.
   - None of Abigail's brothers can sleep.
   - If one of Abigail's brothers makes noise, Abigail cannot sleep.

3. Extend the grammar fragment G_1 to include the ditransitive verb *teaches* and the proper noun *Swahili*. Show how to derive the interpretation for the sentence *Alex teaches Brit Swahili*, which should be TEACHES(ALEX,BRIT,SWAHILI). The grammar need not be in Chomsky Normal Form. For the ditransitive verb, use NP$_1$ and NP$_2$ to indicate the two direct objects.

4. Derive the semantic interpretation for the sentence *Alex likes every dog*, using grammar fragment G_2.

5. Extend the grammar fragment G_2 to handle adjectives, so that the meaning of *an angry dog* is $\lambda P.\exists x \text{DOG}(x) \wedge \text{ANGRY}(x) \wedge P(x)$. Specifically, you should supply the lexical entry for the adjective *angry*, and you should specify the syntactic-semantic productions NP $\rightarrow$ DET NOM, NOM $\rightarrow$ JJ NOM, and NOM $\rightarrow$ NN.

[12] Videos are currently available at `http://yoavartzi.com/tutorial/`
[13] `http://yoavartzi.com/spf`
[14] `https://github.com/percyliang/sempre`

6. Extend your answer to the previous question to cover copula constructions with predicative adjectives, such as *Alex is angry*. The interpretation should be $\text{ANGRY}(\text{ALEX})$. You should add a verb phrase production VP $\rightarrow$ V$_{\text{cop}}$ JJ, and a terminal production V$_{\text{cop}}$ $\rightarrow$ *is*. Show why your grammar extensions result in the correct interpretation.

7. In Figure 12.6 and Figure 12.7, we treat the plurals *shoots* and *leaves* as entities. Revise G_2 so that the interpretation of *Alex eats leaves* is $\forall x.(\text{LEAF}(x) \Rightarrow \text{EATS}(\text{ALEX}, x))$, and show the resulting perceptron update.

8. Statements like *every student eats a pizza* have two possible interpretations, depending on quantifier scope:

$$\forall x \exists y \text{PIZZA}(y) \wedge (\text{STUDENT}(x) \Rightarrow \text{EATS}(x, y)) \tag{12.32}$$

$$\exists y \forall x \text{PIZZA}(y) \wedge (\text{STUDENT}(x) \Rightarrow \text{EATS}(x, y)) \tag{12.33}$$

   Explain why these interpretations really are different, and modify the grammar G_2 so that it can produce both interpretations.

9. Derive Equation 12.27.

10. [todo: not sure this works] Download the GeoQuery data, get some deterministic parser that overgenerates, and try to learn a reranker that selects the correct logical form.

11. In the GeoQuery domain, give a natural language query that has multiple plausible semantic interpretations with the same denotation. List both interpretaions and the denotation.

   Hint: There are many ways to do this, but one approach involves using toponyms (place names) that could plausibly map to several different entities in the model.

Chapter 13

Predicate-argument semantics

In this chapter, we consider more "lightweight" semantic representations. These semantic representations discard some aspects of first-order predicate calculus, but focus on predicate-argument structures. Let's start with an example sentence:

(13.1) Asha gives Boyang a book.

The predicate calculus representation of this sentence would be written,

$$\exists x.\text{BOOK}(x) \wedge \text{GIVE}(\texttt{Asha}, \texttt{Boyang}, x) \tag{13.1}$$

In this representation, we define variable x for the book, and we link the strings *Asha* and *Boyang* to entities `Asha` and `Boyang`. Because the action of giving involves a giver, a recipient, and a gift, the predicate GIVE must take three arguments.

Now suppose we have additional information about the event, such as,

(13.2) Yesterday, Asha reluctantly gave Boyang a book.

One possible to solution is to extend the predicate GIVE to take additional arguments,

$$\exists x.\text{BOOK}(x) \wedge \text{GIVE}(\texttt{Asha}, \texttt{Boyang}, x, \texttt{yesterday}, \texttt{reluctantly}) \tag{13.2}$$

But this is clearly unsatisfactory: *yesterday* and *relunctantly* are optional arguments, and we would need a different version of the GIVE predicate for every possible combination of arguments. **Event semantics** solves this problem by **reifying** the event as an existentially quantified variable e,

$$\exists e, x.\text{GIVE-EVENT}(e) \wedge \text{GIVER}(e, \texttt{Asha}) \wedge \text{GIFT}(e, x) \wedge \text{BOOK}(e, x) \wedge \text{RECIPIENT}(e, \texttt{Boyang})$$
$$\wedge\ \text{TIME}(e, \texttt{Yesterday}) \wedge \text{MANNER}(e, \texttt{reluctantly})$$

In this way, each argument of the event — the giver, the recipient, the gift — can be represented with a relation of its own, linking the argument to the event e. The expression

$\textsc{giver}(e, \texttt{Asha})$ says that $\texttt{Asha}$ plays the **role** of $\textsc{giver}$ in the event. This reformulation nicely handles the problem of optional information such as the time or manner of the event, which are called **adjuncts**. Unlike arguments, adjuncts are not a mandatory part of the relation, but under this representation, they can be expressed with additional logical relations that are conjoined to the semantic interpretation of the setnence. [1]

The event semantic representation can be applied to nested clauses, e.g.,

(13.3) Chris sees Asha pay Boyang.

This is done by using the event variable as an argument:

$$\begin{aligned}\exists e_1, e_2 . \textsc{See-Event}(e_1) \wedge \textsc{seer}(e_1, \texttt{Chris}) \wedge \textsc{sight}(e_1, e_2)\\ \wedge \textsc{Pay-Event}(e_2) \wedge \textsc{payer}(e_2, \texttt{Asha}) \wedge \textsc{payee}(e_2, \texttt{Boyang})\end{aligned} \tag{13.3}$$

As with first-order predicate calculus, the goal of event semantics is to provide a representation that generalizes over many surface forms. Consider the following paraphrases of (13.1):

(13.4) Asha gives a book to Boyang.

(13.5) A book is given to Boyang by Asha.

(13.6) A book is given by Asha to Boyang.

(13.7) The gift of a book from Asha to Boyang ...

All have the same event semantic meaning, given in (13.1). Note that the final example does not include a verb! Events are often introduced by verbs, but not always: in this final example, the noun *gift* introduces the same predicate, with the same accompanying arguments.

Semantic role labeling (SRL) is a relaxed form of semantic parsing, in which each semantic role is filled by a set of tokens from the text itself. This is sometimes called "shallow semantics" because, unlike model-theoretic semantic parsing, role fillers need not be symbolic expressions with denotations in some world model. A semantic role labeling system is required to identify all predicates, and then specify the spans of text that fill each role, when possible. To get a sense of the task, here is a more complicated example:

(13.8) Boyang wants Asha to give him a linguistics book.

In this example, there are two predicates, expressed by the verbs *want* and *give*. Thus, a semantic role labeler should return the following output:

[1]This representation is often called **Neo-Davidsonian event semantics**. The use of existentially-quantified event variables was proposed by Davidson (1967) to handle the issue of optional adjuncts. In "Neo-Davidsonian" semantics, this treatment of adjuncts is extended to mandatory arguments as well (e.g., Parsons, 1990).

- (PREDICATE : *wants*, WANTER : *Boyang*, DESIRE : *Asha to give him a linguistics book*)
- (PREDICATE : *give*, GIVER : *Asha*, RECIPIENT : *him*, GIFT : *a linguistics book*)

In the example, *Boyang* and *him* may refer to the same person, but this would be ignored in semantic role labeling. However, in other predicate-argument representations, such as **Abstract Meaning Representation (AMR)**, such references must be resolved. We will return to AMR in § 13.3, but first, let us further consider the notion of semantic roles.

13.1 Semantic roles

As discussed so far, event semantics requires specifying a number of additional logical relations to link arguments to events: GIVER, RECIPIENT, SEER, SIGHT, etc. Indeed, every predicate requires a set of logical relations to express its own arguments. In contrast, adjuncts such as TIME and MANNER are shared across many types of events. A natural question is whether it is possible to treat mandatory arguments more like adjuncts, by identifying a set of generic argument types that are shared across many event predicates. This can be further motivated by examples involving semantically related verbs:

(13.9) Asha gave Boyang a book.

(13.10) Asha loaned Boyang a book.

(13.11) Asha taught Boyang a lesson.

(13.12) Asha gave Boyang a lesson.

In the first two examples, the roles of Asha, Boyang, and the book are nearly identical. The third example is slightly different, but the fourth example shows that the roles of GIVER and TEACHER can be viewed as related.

One way to think about the relationship between roles such as GIVER and TEACHER is by enumerating the set of properties that an entity typically possesses when it fulfills these roles: givers and teachers are usually animate and "volitional" (meaning that they choose to enter into the action).[2] In contrast, the thing that gets loaned or taught is usually not animate or volitional; furthermore, it is unchanged by the event.

Building on these ideas, **thematic roles** generalize across predicates by leveraging the shared semantic properties of typical role fillers (Fillmore, 1968). For example, in examples (13.9-13.12), Asha plays a similar role in all four sentences, which we will call the **agent**. This reflects a number of shared semantic properties: she is the one who is actively and intentionally performing the action, while Boyang is a more passive participant; the book and the lesson would play a different role, as non-animate participants in the event.

[2]There are always exceptions. For example, in the sentence *The C programming language has taught me a lot about perseverance*, the "teacher" is the *The C programming language*, which is presumably not animate or volitional.

	Asha	*gave*	*Boyang*	*a book*
VerbNet	AGENT		RECIPIENT	THEME
PropBank	ARG0: giver		ARG2: entity given to	ARG1: thing given
FrameNet	DONOR		RECIPIENT	THEME
	Asha	*taught*	*Boyang*	*algebra*
VerbNet	AGENT		RECIPIENT	TOPIC
PropBank	ARG0: teacher		ARG2: student	ARG1: subject
FrameNet	TEACHER		STUDENT	SUBJECT

Figure 13.1: Example semantic annotations according to VerbNet, PropBank, and FrameNet

Let us now consider a few well-known approaches to semantic roles. Example annotations from each of these systems are shown in Figure 13.1.

13.1.1 VerbNet

VerbNet (Kipper-Schuler, 2005) is a lexicon of verbs, and it includes thirty "core" thematic roles played by arguments to these verbs. Here are some example roles, accompanied by their definitions from the VerbNet Guidelines.[3]

- AGENT: "ACTOR in an event who initiates and carries out the event intentionally or consciously, and who exists independently of the event."
- PATIENT: "UNDERGOER in an event that experiences a change of state, location or condition, that is causally involved or directly affected by other participants, and exists independently of the event."
- RECIPIENT: "DESTINATION that is animate"
- THEME: "UNDERGOER that is central to an event or state that does not have control over the way the event occurs, is not structurally changed by the event, and/or is characterized as being in a certain position or condition throughout the state."
- TOPIC: "THEME characterized by information content transferred to another participant."

VerbNet roles are organized in a hierarchy, so that a TOPIC is a type of THEME, which in turn is a type of UNDERGOER, which is a type of PARTICIPANT, the top-level category.

In addition, VerbNet organizes verb senses into a class hierarchy, in which verb senses that have similar meanings are grouped together. Recall from **??** that multiple meanings

[3]`http://verbs.colorado.edu/verb-index/VerbNet_Guidelines.pdf`

of the same word are called **senses**, and that WordNet identifies senses for many English words. VerbNet builds on WordNet, so that verb classes are identified by the WordNet senses of the verbs that they contain. For example, the verb class `give-13.1` includes the first WordNet sense of *loan* and the second WordNet sense of *lend*.

Each VerbNet class or subclass takes a set of thematic roles. For example, `give-13.1` takes arguments with the thematic roles of AGENT, THEME, and RECIPIENT;[4] the predicate TEACH takes arguments with the thematic roles AGENT, TOPIC, RECIPIENT, and SOURCE.[5] So according to VerbNet, *Asha* and *Boyang* play the roles of AGENT and RECIPIENT in the sentences,

(13.13) Asha gave Boyang a book.

(13.14) Asha taught Boyang algebra.

The *book* and *algebra* are both THEMES, but *algebra* is a subcategory of THEME — a TOPIC — because it consists of information content that is given to the receiver.

13.1.2 Proto-roles and PropBank

Detailed thematic role inventories of the sort used in VerbNet are not universally accepted. For example, (Dowty, 1991, pp. 547) notes that "Linguists have often found it hard to agree on, and to motivate, the location of the boundary between role types." He argues that a solid distinction can be identified between just two **proto-roles**, which have a number of distinguishing characteristics:

- PROTO-AGENT: volitional involvement in the event or state; sentience and/or perception; causing an event or change of state in another participant; movement; exists independently of the event.
- PROTO-PATIENT: undergoes change of state; causally affected by another participant; stationary relative to the movement of another participant; does not exist independently of the event.[6]

In the examples in Figure 13.1, Asha has most of the proto-agent properties: in giving the book to Boyang, she is acting volitionally (as opposed to *Boyang got a book from Asha*, in which it is not clear whether Asha gave up the book willingly); she is sentient; she causes a change of state in Boyang; she exists independently of the event. Boyang has some proto-agent properties (for example, he is sentient and exists independently of the

[4] `https://verbs.colorado.edu/verb-index/vn/give-13.1.php`

[5] `https://verbs.colorado.edu/verb-index/vn/transfer_mesg-37.1.1.php`

[6] Reisinger et al. (2015) ask crowd workers to annotate these properties directly, finding that annotators tend to agree on the properties of each argument. They also find that in English, arguments having more proto-agent properties tend to appear in subject position, while arguments with more proto-patient properties appear in object position.

TMP	time	*Boyang ate a bagel* [$_{\text{AM-TMP}}$ *yesterday*].
LOC	location	*Asha studies in* [$_{\text{AM-LOC}}$ *Stuttgart*]
MOD	modal verb	*Asha* [$_{\text{AM-MOD}}$ *will*] *study in Stuttgart*
ADV	general purpose	[$_{\text{AM-ADV}}$ *Luckily*], *Asha knew algebra.*
MNR	manner	*Asha ate* [$_{\text{AM-MNR}}$ *aggressively*].
DIS	discourse connective	[$_{\text{AM-DIS}}$ *However*], *Asha prefers algebra.*
PRP	purpose	*Barry studied* [$_{\text{AM-PRP}}$ *to pass the bar*].
DIR	direction	*Workers dumped burlap sacks* [$_{\text{AM-DIR}}$ *into a bin*].
NEG	negation	*Asha does* [$_{\text{AM-NEG}}$ *not*] *know algebra.*
EXT	extent	*Prices increased* [$_{\text{AM-EXT}}$ *4%*].
CAU	cause	*Asha returned the book* [$_{\text{AM-CAU}}$ *because it was overdue*].

Table 13.1: PropBank adjuncts (Palmer et al., 2005), sorted by frequency in the corpus

event), but he also some proto-patient properties: he is the one who is causally affected and who undergoes change of state in both cases. The book that Asha gives Boyang has fewer still of the proto-agent properties — it is not volitional or sentient, and it has no causal role — but it also has few proto-patient properties, as it does not undergo change of state and is not stationary.

The **Proposition Bank**, or PropBank (Palmer et al., 2005), builds on this basic agent-patient distinction, as a middle ground between generic thematic roles and predicate-specific "deep roles." Each verb is linked to a list of numbered arguments, with ARG0 as the proto-agent and ARG1 as the proto-patient. Additional numbered arguments are verb-specific. For example, for the predicate TEACH,[7] the arguments are:

- ARG0: the teacher
- ARG1: the subject
- ARG2: the student(s)

Verbs may have any number of arguments: for example, WANT and GET have five, while EAT has only ARG0 and ARG1. In addition to the semantic arguments found in the frame files, roughly a dozen general-purpose **adjuncts** may be used in combination with any verb. These are shown in Table 13.1.

PropBank-style semantic role labeling is annotated over the entire Penn Treebank. This annotation includes the sense of each verbal predicate, as well as the argument spans.

[7] http://verbs.colorado.edu/propbank/framesets-english-aliases/teach.html

13.1.3 FrameNet

Semantic **frames** are descriptions of situations or events. Frames may be **evoked** by one of their **lexical units** (often a verb, but not always), and they include some number of **frame elements**, which are like roles (Fillmore, 1976). For example, the act of teaching is a frame, and can be evoked by the verb *taught*; the associated frame elements include the teacher, the student(s), and the subject being taught. Frame semantics has played a significant role in the history of artificial intelligence, in the work of Minsky (1974) and Schank and Abelson (1977). In natural language processing, the theory of frame semantics has been implemented in **FrameNet** (Fillmore and Baker, 2009), which consists of a lexicon of roughly 1000 frames, and a corpus of more than 200,000 "exemplar sentences," in which the frames and their elements are annotated.[8]

Rather than seeking to link semantic roles such as TEACHER and GIVER into thematic roles such as AGENT, FrameNet aggressively groups verbs into frames, and links semantically-related roles across frames. For example, the following two sentences would be annotated identically in FrameNet:

(13.15) Asha taught Boyang algebra.

(13.16) Boyang learned algebra from Asha.

This is because *teach* and *learn* are both lexical units in the EDUCATION_TEACHING frame. Furthermore, roles can be shared even when the frames are distinct, as in the following two examples:

(13.17) Asha gave Boyang a book.

(13.18) Boyang got a book from Asha.

The GIVING and GETTING frames both have RECIPIENT and THEME elements, so Boyang and the book would play the same role. Asha's role is different: she is the DONOR in the GIVING frame, and the SOURCE in the GETTING frame. FrameNet makes extensive use of multiple inheritance to share information across frames and frame elements: for example, the COMMERCE_SELL and LENDING frames inherit from GIVING frame.

13.2 Semantic role labeling

The task of semantic role labeling is to identify the parts of the sentence comprising the semantic roles. In English, this task is typically performed on the PropBank corpus, with the goal of producing outputs in the following form:

(13.19) [$_{\text{ARG0}}$ Asha] [$_{\text{GIVE.01}}$ gave] [$_{\text{ARG2}}$ Boyang's mom] [$_{\text{ARG1}}$ a book] [$_{\text{AM-TMP}}$ yesterday].

[8]These statistics are accurate at the time of this writing, in 2017. Current details can be found at the website, `https://framenet.icsi.berkeley.edu/`

Note that a single sentence may have multiple verbs, and therefore a given word may be part of multiple role-fillers:

(13.20) [$_{\text{ARG0}}$ Asha] [$_{\text{WANT.01}}$ wanted] [$_{\text{ARG1}}$ Boyang to give her the book].
Asha wanted [$_{\text{ARG0}}$ Boyang] [$_{\text{GIVE.01}}$ to give] [$_{\text{ARG2}}$ her] [$_{\text{ARG1}}$ the book].

13.2.1 Semantic role labeling as classification

PropBank is annotated on the Penn Treebank, and annotators used phrasal constituents (chapter 9) to fill the roles. Therefore SRL can be viewed as the task of assigning to each phrase a label from the set $\mathcal{R} = \{\varnothing, \text{PRED}, \text{ARG0}, \text{ARG1}, \text{ARG2}, \ldots, \text{AM-LOC}, \text{AM-TMP}, \ldots\}$, where $\varnothing$ indicates that the phrase plays no role, and PRED indicates that it is the verbal predicate. If we treat semantic role labeling as a classification problem, we obtain the following functional form:

$$\hat{y}_{(i,j)} = \underset{y}{\operatorname{argmax}}\, \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, y, i, j, \rho, \tau), \tag{13.4}$$

where,

- (i, j) indicates the span of a phrasal constituent $(w_i, w_{i+1}, \ldots, w_{j-1})$;[9]
- $\boldsymbol{w}$ represents the sentence as a sequence of tokens;
- ρ is the index of the predicate verb in $\boldsymbol{w}$;
- τ is the structure of the phrasal constituent parse of $\boldsymbol{w}$.

Table 13.2 shows the features used in the seminal paper on FrameNet semantic role labeling by Gildea and Jurafsky (2002). By 2005 there were several systems for PropBank semantic role labeling, and their approaches and feature sets are summarized by Carreras and Màrquez (2005). Typical features include: the phrase type, head word, part-of-speech, boundaries, and neighbors of the proposed argument $\boldsymbol{w}_{i:j}$; the word, lemma, part-of-speech, and voice of the verb w_ρ (active or passive), as well as features relating to its frameset; the distance and path between the verb and the proposed argument. In this way, semantic role labeling systems are high-level "consumers" in the NLP stack, using features produced from lower-level components such as part-of-speech taggers and parsers. More comprehensive and contemporary feature sets are enumerated by Das et al. (2014) and Täckström et al. (2015).

[9]PropBank roles can also be filled by **split constituents**, which are discontinuous spans of text. This situation most frequently in reported speech, e.g. [$_{\text{ARG1}}$ *By addressing these problems*], *Mr. Maxwell said,* [$_{\text{ARG1}}$ *the new funds have become extremely attractive.*] (example adapted from Palmer et al., 2005). This issue is typically addressed by defining "continuation arguments", e.g. C-ARG1, which refers to the continuation of ARG1 after the split.

Predicate lemma and POS tag	The lemma of the predicate verb and its part-of-speech tag
Voice	Whether the predicate is in active or passive voice, as determined by a set of syntactic patterns for identifying passive voice constructions
Phrase type	The constituent phrase type for the proposed argument in the parse tree, e.g. NP, PP
Headword and POS tag	The head word of the proposed argument and its POS tag, identified using the Collins (1997) rules
Position	Whether the proposed argument comes before or after the predicate in the sentence
Syntactic path	The set of steps on the parse tree from the proposed argument to the predicate (described in detail in the text)
Subcategorization	The syntactic production from the first branching node above the predicate. For example, in Figure 13.2, the subcategorization feature around *taught* would be VP $\rightarrow$ VBD NP PP.

Table 13.2: Features used in semantic role labeling by Gildea and Jurafsky (2002).

A particularly powerful class of features relate to the **syntactic path** between the argument and the predicate. These features capture the sequence of moves required to get from the argument to the verb by traversing the phrasal constituent parse of the sentence. The idea of these features is to capture syntactic regularities in how various arguments are realized. Syntactic path features are best illustrated by example, using the parse tree in Figure 13.2:

- The path from *Asha* to the verb *taught* is NNP↑NP↑S↓VP↓VBD. The first part of the path, NNP↑NP↑S, means that we must travel up the parse tree from the NNP tag (proper noun) to the S (sentence) constituent. The second part of the path, S↓VP↓VBD, means that we reach the verb by producing a VP (verb phrase) from the S constituent, and then by producing a VBD (past tense verb). This feature is consistent with *Asha* being in subject position, since the path includes the sentence root S.
- The path from *the class* to the verb is NP↑VP↓VBD. This is consistent with *the class* being in object position, since the path passes through the VP node that dominates the verb *taught*.

Because there are many possible path features, it can also be helpful to look at smaller parts: for example, the upward and downward parts can be treated as separate features; another feature might consider whether S appears anywhere in the path.

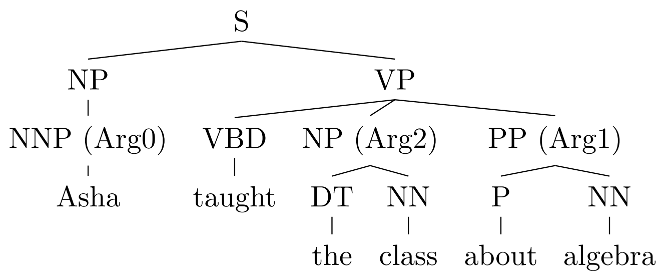


Figure 13.2: Semantic role labeling is often performed on the parse tree for a sentence, labeling individual constituents. [todo: check arg1; show arrows for path features]

Rather than using the constituent parse, it is also possible to build features from the **dependency path** between the head word of each argument and the verb (Pradhan et al., 2005). Using the Universal Dependency part-of-speech tagset and dependency relations (Nivre et al., 2016), the dependency path from *Asha* to *taught* is $\text{PROPN} \underset{\text{NSUBJ}}{\leftarrow} \text{VERB}$, because *taught* is the head of a relation of type NSUBJ with *Asha*. Similarly, the dependency path from *class* to *taught* is $\text{NOUN} \underset{\text{DOBJ}}{\leftarrow} \text{VERB}$, because *class* heads the noun phrase that is a direct object of *taught*. A more interesting example is *Asha tried to teach the class*, where the path from *Asha* to *tried* is $\text{PROPN} \underset{\text{NSUBJ}}{\leftarrow} \text{VERB} \underset{\text{XCOMP}}{\rightarrow} \text{VERB}$. The right-facing arrow in second relation indicates that *tried* is the head of its XCOMP relation with *teach*.

13.2.2 Semantic role labeling as constrained optimization

A potential problem with treating SRL as a classification problem is that there are a number of sentence-level **constraints**, which a classifier might violate.

- For a given verb, there can be only one argument of each type (ARG0, ARG1, etc.)
- Arguments cannot overlap. This problem arises when we are labeling the phrases in a constituent parse tree, as shown in Figure 13.2: if we label the PP *about algebra* as an argument or adjunct, then its children *about* and *algebra* must be labeled as $\varnothing$. The same constraint also applies to the syntactic ancestors of this phrase.

These constraints can be viewed as introducing dependencies across labeling decisions. In structure prediction problems such as sequence labeling and parsing, such dependencies are usually handled by defining additional features over the entire structure, $\boldsymbol{y}$. Efficient inference requires that the global features have a local decomposition that enables dynamic programming: for example, in sequence labeling, the features over $\boldsymbol{y}$ were decomposed into features over pairs of adjacent tags, permitting the application of the Viterbi algorithm for inference. Unfortunately, the constraints that arise in semantic role labeling are less amenable to local decomposition — particularly the constraint that each argument is used only once in the sentence.[10] We therefore consider **constrained optimization** as an alternative solution.

Let the **feasible set** $\mathcal{C}(\tau)$ refer to all labelings that obey the constraints introduced by the parse τ. We can reformulate the semantic role labeling problem as a constrained optimization,

$$\begin{aligned} \max_{\boldsymbol{y}} \quad & \sum_{(i,j)\in\tau} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, y_{(i,j)}, i, j, \rho, \tau) \\ s.t. \quad & \boldsymbol{y} \in \mathcal{C}(\tau). \end{aligned} \tag{13.5}$$

In this formulation, the objective (shown on the first line) is a separable function of each individual labeling decision, but the constraints (shown on the second line) apply to the overall labeling. The sum $\sum_{(i,j)\in\tau}$ indicates that we are summing over all constituent spans in the parse τ. The expression $s.t.$ in the second line means that we maximize the objective *subject to* the constraints that appear to the right.

Integer linear programming

A number of practical algorithms exist for restricted forms of constrained optimization. One such restricted form is **integer linear programming**, in which we optimize a linear objective function over integer variables, with linear constraints. To formulate SRL as an integer linear program, we begin by rewriting the labels as a set of binary variables $\boldsymbol{z} = \{z_{i,j,r}\}$, where,

$$z_{i,j,r} = \begin{cases} 1, & y_{(i,j)} = r \\ 0, & \text{otherwise.} \end{cases} \tag{13.6}$$

Thus, the variables $\boldsymbol{z}$ are a binarized version of the semantic role labeling $\boldsymbol{y}$.

Objective Next, we restrict the objective to be a linear function of $\boldsymbol{z}$. We begin with the feature function, $\boldsymbol{f}(\boldsymbol{w}, y_{(i,j)}, i, j, \rho, \tau)$. Such features are typically logical conjunctions involving the label $y_{(i,j)}$. For example:

$$f_j(\boldsymbol{w}, y_{(i,j)}, i, j, \rho, \tau) = \begin{cases} 1, & y_{(i,j)} = \textsc{Arg1} \wedge w_i = \textit{the} \\ 0, & \text{otherwise.} \end{cases} \tag{13.7}$$

This feature is an indicator that takes the value 1 for constituents that are labeled ARG1 and begin with the word *the*. If all features are conjunctions with the label, then the feature

[10]Dynamic programming solutions have been proposed by Tromble and Eisner (2006) and Täckström et al. (2015), but they involves creating a trellis structure whose size is exponential in the number of labels.

function can be rewritten,

$$\boldsymbol{f}(\boldsymbol{w}, y_{(i,j)}, i, j, \rho, \tau) = \sum_{r \in \mathcal{R}} z_{i,j,r} \times \boldsymbol{g}(\boldsymbol{w}, i, j, r, \rho, \tau), \tag{13.8}$$

where $\mathcal{R}$ is the set of all possible labels, $\{\text{ARG0}, \text{ARG1}, \ldots, \text{AM-LOC}, \ldots, \varnothing\}$. With this change in notation, we can now rewrite the objective,

$$\begin{aligned}
\sum_{(i,j)\in\tau} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{w}, y_{(i,j)}, i, j, \rho, \tau) &= \sum_{(i,j)\in\tau} \boldsymbol{\theta} \cdot \left(\sum_{r\in\mathcal{R}} z_{i,j,r} \times \boldsymbol{g}(\boldsymbol{w}, i, j, r, \rho, \tau) \right) && (13.9)\\
&= \sum_{(i,j)\in\tau} \sum_{r\in\mathcal{R}} z_{i,j,r} \left(\boldsymbol{\theta} \cdot \boldsymbol{g}(\boldsymbol{w}, i, j, r, \rho, \tau) \right) && (13.10)\\
&= \sum_{(i,j)\in\tau} \sum_{r\in\mathcal{R}} z_{i,j,r} \psi_{i,j,r}, && (13.11)
\end{aligned}$$

where $\psi_{i,j,r} \triangleq \boldsymbol{\theta} \cdot \boldsymbol{g}(\boldsymbol{w}, i, j, r, \rho, \tau)$. This objective is clearly a linear function of the variables $\boldsymbol{z} = \{z_{i,j,r}\}$.

Constraints Integer linear programming permits linear inequality constraints, of the general form $\mathbf{A}\boldsymbol{z} \leq \boldsymbol{b}$, where the parameters $\mathbf{A}$ and $\boldsymbol{b}$ define the constraints. To make this more concrete, let's start with the constraint that each non-null role type can occur only once in a sentence. This constraint can be written,

$$\forall r \neq \varnothing, \qquad \sum_{(i,j)\in\tau} z_{i,j,r} \leq 1. \tag{13.12}$$

Recall that $z_{i,j,r} = 1$ if and only if the span (i, j) has label r; this constraint says that for each possible label $r \neq \varnothing$, there can be at most one (i, j) such that $z_{i,j,r} = 1$. This constraint can be written in the form $\mathbf{A}\boldsymbol{z} \leq \boldsymbol{b}$, as you will find if you complete the exercises at the end of the chapter.

Now consider the constraint that labels cannot overlap. Let the function $\pi_\tau(i, j) = \{(i', j')\}$ indicate the set of constituents that are ancestors or descendents of (i, j) in the parse τ. For any (i, j) such that $y_{i,j} \neq \varnothing$, the non-overlapping constraint means that all of its ancestors and descendents (i', j') must be labeled as a non-argument, $y_{i',j'} = \varnothing$. We can write this as a set of linear constraints,

$$\forall (i, j) \in \tau, \quad \sum_{r \neq \varnothing} \left(z_{i,j,r} + \sum_{(i',j')\in\pi_\tau(i,j)} z_{i',j',r} \right) \leq 1. \tag{13.13}$$

We can therefore rewrite the semantic role labeling problem as the following integer linear program,

$$\max_{\boldsymbol{z} \in \{0,1\}^{|\tau|}} \quad \sum_{(i,j)\in\tau} \sum_{r \in \mathcal{R}} z_{i,j,r} \psi_{i,j,r} \tag{13.14}$$

$$s.t. \quad \forall r \neq \varnothing, \quad \sum_{(i,j)\in\tau} z_{i,j,r} \leq 1. \tag{13.15}$$

$$\forall (i,j) \in \tau, \quad \sum_{r \neq \varnothing} \left(z_{i,j,r} + \sum_{(i',j') \in \pi_\tau(i,j)} z_{i',j',r} \right) \leq 1. \tag{13.16}$$

The effectiveness of integer linear programming for semantic role labeling was first demonstrated by Punyakanok et al. (2008).

Learning with constraints Learning can be performed in the context of constrained optimization using the usual perceptron or large-margin classification updates. Because constrained inference is generally more time-consuming, a key question is whether it is necessary to apply the constraints during learning. Chang et al. (2008) find that better performance can be obtained by learning *without* constraints, and then applying constraints only when using the trained model to predict semantic roles for unseen data.

How important are the constraints? Das et al. (2014) find that an unconstrained, classification-based method performs nearly as well as constrained optimization for FrameNet parsing: while it commits many violations of the "no-overlap" constraint, the overall F_1 score is less than one point worse than the score at the constrained optimum. Similar results are obtained for PropBank semantic role labeling by Punyakanok et al. (2008). He et al. (2017) find that constrained inference makes a bigger impact if the constraints are based on manually-labeled "gold" syntactic parses. This implies that errors from the syntactic parser may limit the effectiveness of the constraints. Punyakanok et al. (2008) hedge against parser error by including constituents from several different parsers; any constituent can be selected from any parse, and additional constraints ensure that overlapping constituents are not selected.

Implementation Integer linear programming solvers such as `glpk`,[11] `cplex`,[12] and `Gurobi`[13] allow inequality constraints to be expressed directly in the problem definition, rather than in the matrix form $\mathbf{A}\boldsymbol{z} \leq \boldsymbol{b}$. The time complexity of integer linear programming is theoretically exponential in the number of variables $|\boldsymbol{z}|$, but in practice these off-the-shelf solvers

[11]https://www.gnu.org/software/glpk/
[12]https://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/
[13]http://www.gurobi.com/

obtain good solutions efficiently. Das et al. (2014) report that the `cplex` solver requires 43 seconds to perform inference on the FrameNet test set, which contains 4,458 predicates.

Recent work has shown that many constrained optimization problems in natural language processing can be solved in a highly parallelized fashion, using optimization techniques such as **dual decomposition**, which are capable of exploiting the underlying problem structure (Rush et al., 2010). Das et al. (2014) apply this technique to FrameNet semantic role labeling, obtaining an order-of-magnitude speedup over `cplex`.

13.2.3 Neural semantic role labeling

Neural network approaches to SRL have tended to treat it as a sequence labeling task, using a labeling scheme such as the **BIO notation**, which we previously saw in named entity recognition (§ 7.4). In this notation, the first token in a span of type ARG1 is labeled B-ARG1; all remaining tokens in the span are **inside**, and are therefore labeled I-ARG1. Tokens outside any argument are labeled O. For example:

(13.21)	*Asha*	*taught*	*Boyang*	*'s*	*mom*	*about*	*algebra*
	B-ARG0	PRED	B-ARG2	I-ARG2	I-ARG2	B-ARG1	I-ARG1

We now consider two classes of neural networks that can learn to produce such labelings.

Convolutional neural networks One of the first applications of **convolutional neural networks** (§ 2.7.2) to natural language processing was the task of classifying semantic roles. Collobert et al. (2011b) treat the task as a classification problem, using information gathered from across the sentence to compute the label for each token. For example, suppose our goal is to tag the role of word m with respect to verb v; then for word n, we compute the discrete feature vector, $\boldsymbol{f}(\boldsymbol{w}, n, v, m)$, which would include the lower-case word w_m, and the distances $m - n$ and $m - v$. These features are then used as the inputs to a nonlinear prediction model, as follows:

- Each discrete feature is associated with a dense vector embedding, and these embeddings are concatenated, resulting in a dense vector $\boldsymbol{x}_m^{(0)}$. The horizontal concatenation of the dense embeddings for all words in the sentence is $\mathbf{X}^{(0)} = [\boldsymbol{x}_0^{(0)}, \boldsymbol{x}_1^{(0)}, \ldots, \boldsymbol{x}_M^{(0)}]$.
- Next, a convolutional operation is applied to merge information across words, $\mathbf{X}^{(1)} = \mathbf{C}\mathbf{X}^{(0)}$. Thus, $\boldsymbol{x}_m^{(1)}$ contains information about the word w_m, but also about its near neighbors. (Special padding vectors are included on the left and right ends of the matrix $\mathbf{X}^{(0)}$ before convolution.)
- To convert the matrix $\mathbf{X}^{(1)}$ back to a vector $\boldsymbol{z}^{(1)}$, Collobert *et al.* apply **max pooling**. We will write $\boldsymbol{z} = \text{MaxPool}(\mathbf{X})$ to indicate that each $z_j = \max_m(x_{0,j}^{(1)}, x_{1,j}^{(1)}, \ldots, x_{M,j}^{(1)})$.

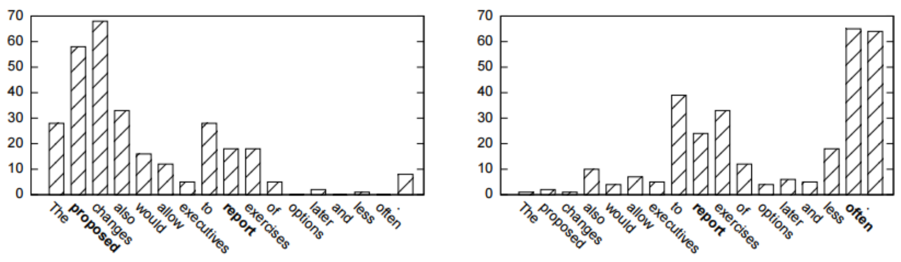


Figure 13.3: Number of features chosen at each word position by the max pooling operation, for tagging the words *proposed* (left) and *often* (right). Figure reprinted from Collobert et al. (2011b) [todo: ask for permission]

- The vector $\boldsymbol{z}^{(1)}$ is then passed through several feedforward layers, $\boldsymbol{z}^{(i)} = \boldsymbol{g}(\boldsymbol{\Theta}^{(i)}\boldsymbol{z}^{(i-1)})$, where $\boldsymbol{\Theta}^{(i)}$ is a matrix of weights and $\boldsymbol{g}$ is an elementwise nonlinear transformation.[14]
- At the output layer $\boldsymbol{z}^{(K)}$ is used to make a prediction, $\hat{y} = \text{argmax}_y \, \boldsymbol{\Theta}^{(y)}\boldsymbol{z}^{(K)}$. [todo: consider making this a figure/algorithm]

Collobert et al. (2011b) apply this tagging model without regard to constraints, so in principle it could produce labelings that include each argument multiple times. The parameters of the model include the word and feature embeddings that constitute $\mathbf{X}^{(0)}$, the convolution matrix $\mathbf{C}$, the feedforward weight matrices $\boldsymbol{\Theta}^{(i)}$, and the prediction weights $\boldsymbol{\Theta}^{(y)}$. Each of these parameters is estimated by backpropagated stochastic gradient descent (see § 5.3.1). Collobert et al. (2011b) emphasize that **multi-task learning** was essential to get good performance: they train the word embeddings not only to accurately predict PropBank labels, but also to assign a high likelihood to a large corpus of unlabeled data. A more contemporary approach would be to use **pre-trained word embeddings**, which have already been trained to predict words in context, thereby avoiding the cost of jointly training across a large unlabeled dataset.

A key aspect of convolutional neural networks is the use of **pooling** operations, which combine information across a variable-length sequence of vectors into a single vector or

[14]Collobert *et al.* use a piecewise linear **hard tanh** function for nonlinear transformations,

$$g(x) = \begin{cases} -1, & x < -1 \\ x, & -1 \leq x \leq 1 \\ 1, & x > 1. \end{cases} \tag{13.17}$$

An advantage of this function is that the gradient is easy and fast to compute, making training faster. More recent work has emphasized the **rectified linear unit (ReLU)**, $g(x) = \max(x, 0)$. This function, which is described in chapter 5), also has similar advantages to hard tanh, but avoids saturation because it has a non-zero gradient for large values of x (Salinas and Abbott, 1996; Glorot et al., 2011).

matrix. Max pooling is widely used in natural language processing applications, because it enables each element in the vector $\boldsymbol{z}$ to take information from across a sentence or other sequence of text. Figure 13.3 shows the number of "features" taken from each word in a sentence — that is, how often the max operation chooses an element from each word. In each case, the pooling operation emphasizes the word to be tagged, its neighbors, and also the main verb *report*.

Recurrent neural networks An alternative neural approach to semantic role labeling is to use **recurrent neural network** models, such as **long short-term memories** (LSTMs; see § 6.5.4 to review how these models are applied to tagging tasks). Zhou and Xu (2015) apply a bidirectional multilayer LSTM to PropBank semantic role labeling. In this model, each bidirectional LSTM serves as input for another, higher-level bidirectional LSTM, allowing complex non-linear transformations of the original input embeddings, $\mathbf{X} = [\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_M]$. The hidden state of the final LSTM is $\mathbf{Z}^{(K)} = [\boldsymbol{z}_1^{(K)}, \boldsymbol{z}_2^{(K)}, \ldots, \boldsymbol{z}_M^{(K)}]$. The "emission" score for each tag $Y_m = y$ is equal to the inner product $\boldsymbol{\theta}_y \cdot \boldsymbol{z}_m^{(K)}$, and there is also a transition score for each pair of adjacent tags. The complete model can be written,

$$\mathbf{Z}^{(1)} = \text{BiLSTM}(\mathbf{X}) \tag{13.18}$$

$$\mathbf{Z}^{(i)} = \text{BiLSTM}(\mathbf{Z}^{(i-1)}) \tag{13.19}$$

$$\hat{\boldsymbol{y}} = \underset{\boldsymbol{y}}{\operatorname{argmax}} \sum_{m-1}^{M} \boldsymbol{\Theta}^{(y)} \boldsymbol{z}_m^{(K)} + \psi_{y_{m-1}, y_m}. \tag{13.20}$$

Note that the final step maximizes over the entire labeling $\boldsymbol{y}$, and includes a score for each tag transition ψ_{y_{m-1}, y_m}. This combination of LSTM and pairwise potentials on tags is an example of an **LSTM-CRF**. The maximization over $\boldsymbol{y}$ is performed by the Viterbi algorithm.

This model strongly outperformed alternative approaches at the time, including constrained decoding and convolutional neural networks. More recent work has combined recurrent neural network models with constrained decoding, using the A^* search algorithm to search over labelings that are feasible with respect to the constraints (He et al., 2017). This yields small improvements over the method of Zhou and Xu (2015). He et al. (2017) obtain larger improvements by creating an **ensemble** of SRL systems, each trained on an 80% subsample of the corpus. The average prediction across this ensemble is more robust than any individual model.

13.3 Abstract Meaning Representation

Semantic role labeling transforms the task of semantic parsing to a labeling task. Consider the sentence,

```
(w / want-01
  :ARG0 (b / boy)
  :ARG1 (g / go-02
          :ARG0 b))
```

$$\begin{tikzcd}
& \texttt{w / want-01} \arrow[dl, "\textsc{Arg0}"'] \arrow[dr, "\textsc{Arg1}"] & \\
\texttt{b / boy} & & \texttt{g / go-02} \arrow[ll, "\textsc{Arg1}"]
\end{tikzcd}$$

Figure 13.4: Two views of the AMR representation for the sentence *The boy wants to go.*

(13.22) The boy wants to go.

The PropBank semantic role labeling analysis is:

- (PREDICATE : *wants*, ARG0 : *the boy*, ARG1 : *to go*)
- (PREDICATE : *go*, ARG1 : *the boy*)

The **Abstract Meaning Representation (AMR)** unifies this analysis into a graph structure, in which each node is a **variable**, and each edge indicates a **concept** (Banarescu et al., 2013). This can be written in two ways, as shown in Figure 13.4. On the left is the PENMAN notation (Matthiessen and Bateman, 1991), in which each set of parentheses introduces a variable. Each variable is an **instance** of a concept, which is indicated with the slash notation: for example, `w / want-01` indicates that the variable `w` is an instance of the concept `want-01`, which in turn refers to the PropBank frame for the first sense of the verb *want*. Relations are introduced with colons: for example, `:arg0 (b / boy)` indicates a relation of type `arg0` with the newly-introduced variable `b`. Variables can be reused, so that when the variable `b` appears again as an argument to `g`, it is understood to refer to the same boy in both cases. This arrangement is indicated compactly in the graph structure on the right, with edges indicating concepts.

AMR differs from PropBank-style semantic role labeling in a few key ways. First, it reifies each entity as a variable: for example, the *boy* in (13.22) is reified in the variable `b`, which is reused as ARG0 in its relationship with `w / want-01`, and as ARG1 in its relationship with `g / go-02`. Reifying entities as variables also makes it possible to represent the substructure of noun phrases more explicitly. For example, *Asha borrowed the algebra book* would be represented as:

```
(b / borrow-01
   :ARG0 (p / person
           :name (n / name
                    :op1 "Asha"))
   :ARG1 (b2 / book
           :topic (a / algebra)))
```

This indicates that the variable `p` is a person, whose `name` is the variable `n`; that name has one token, the string *Asha*. Similarly, the variable `b2` is a book, and the `topic` of `b2` is a variable a whose type is algebra. The relations name and topic are examples of **non-core roles**, which are similar to adjunct modifiers in PropBank. However, AMR's inventory is more extensive, including more than 70 non-core roles, such as negation, time, manner, frequency, and location. Lists and sequences — such as the list of tokens in a name — are described using the roles op1, op2, etc.

Another key feature of AMR is that a semantic predicate can be introduced by any syntactic element. Consider the following examples, from Banarescu et al. (2013):

(13.23) The boy destroyed the room.

(13.24) the destruction of the room by the boy . . .

(13.25) the boy's destruction of the room . . .

All these examples have the same semantics in AMR,

```
(d / destroy-01
   :arg0 (b / boy)
   :arg1 (r / room))
```

Note that the noun *destruction* is linked to the verb *destroy*, which is captured by the PropBank frame destroy-01. This can happen with adjectives as well: in the phrase *the attractive spy*, the adjective *attractive* is linked to the PropBank frame attract-01:

```
(s / spy
   :arg0-of (a / attract-01))
```

In this example, arg0-of is an **inverse relation**, indicating that s is the arg0 of the predicate a. Inverse relations make it possible for all AMR parses to have a single root concept, which should be the **focus** of the utterance.

There are a number of other important linguistic issues in the design of AMR, which are summarized in the original paper (Banarescu et al., 2013) and the tutorial slides by Schneider et al. (2015). While AMR goes farther than semantic role labeling, it does not link semantically-related frames such as buy/sell (as FrameNet does), does not handle quantification (as first-order predicate calculus does), and makes no attempt to handle noun number and verb tense (as PropBank does). A recent survey by Abend and Rappoport (2017) situates AMR with respect to several other semantic representation schemes.

13.3.1 AMR Parsing

Abstract Meaning Representation is not a labeling of the original text — unlike PropBank semantic role labeling, and most of the other tagging and parsing tasks that we have encountered thus far. The AMR for a given sentence may include multiple concepts for single words in the sentence: as we have seen, the sentence *Asha likes algebra* contains both `person` and `name` concepts for the word *Asha*. Conversely, words in the sentence may not appear in the AMR: in *Boyang made a tour of campus*, the **light verb** *make* would not appear in the AMR, which would instead be rooted on the predicate `tour`. As a result, AMR is difficult to parse, and even evaluating AMR parsing involves considerable algorithmic complexity (Cai and Yates, 2013).

A further complexity is that AMR labeled datasets do not explicitly show the **alignment** between the AMR annotation and the words in the sentence. For example, the link between the word *wants* and the concept `want-01` is not annotated. To acquire training data for learning-based parsers, it is therefore necessary to first perform an alignment between the training sentences and their AMR parses. Flanigan et al. (2014) introduce a rule-based parser, which links text to concepts through a series of increasingly high-recall steps.

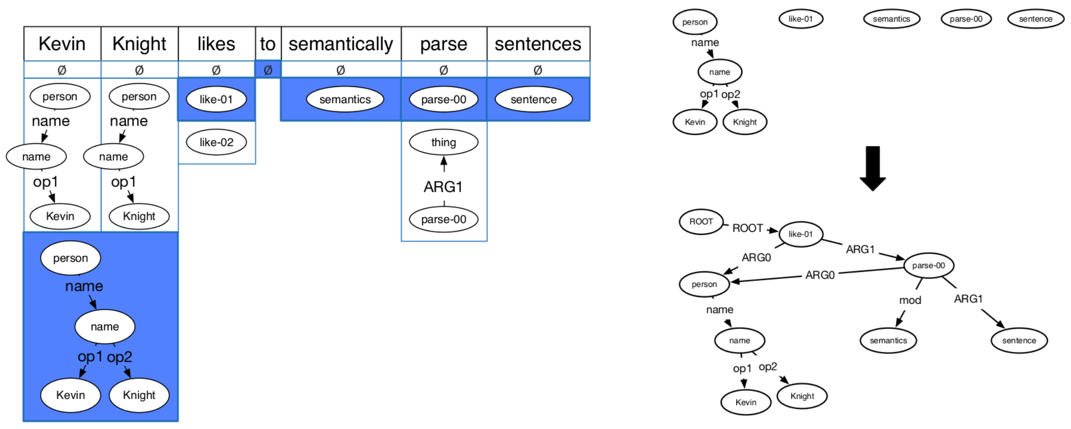
Graph-based parsing One family of approaches to AMR parsing is similar to the graph-based methods that we encountered in syntactic dependency parsing (chapter 11). For these systems (Flanigan et al., 2014), parsing is a two-step process:

1. **Concept identification** (Figure 13.5a). This involves constructing concept subgraphs for individual words or spans of adjacent words. For example, in the sentence, *Asha likes algebra*, we would hope to identify the minimal subtree including just the concept `like-01` for the word *like*, and the subtree `(p / person :name (n / name :op1 Asha))` for the word *Asha*.

2. **Relation identification** (Figure 13.5b). This involves building a directed graph over the concepts, where the edges are labeled by the relation type. AMR imposes a number of constraints on the graph: all concepts must be included, the graph must be **connected** (there must be a path between every pair of nodes in the undirected version of the graph), and every node must have at most one outgoing edge of each type.

Both of these problems are solved by structure prediction. Concept identification requires simultaneously segmenting the text into spans, and labeling each span with a graph fragment containing one or more concepts. This is done by computing a set of features for each candidate span $\boldsymbol{s}$ and concept labeling $\boldsymbol{c}$, and then returning the labeling with the highest overall score.

Relation identification can be formulated as search for the maximum spanning subgraph, under a set of constraints. Each labeled edge has a score, which is computed from features of the concepts. We then search for the set of labeled edges that maximizes the sum of these scores, under the constraint that the resulting graph is well-formed AMR. § 13.2.2 described how constrained optimization can be used for semantic role labeling; similar techniques have been applied to AMR relation identification (Flanigan et al., 2014).



(a) Concept identification (b) Relation identification

Figure 13.5: Subtasks for Abstract Meaning Representation parsing, from Schneider et al. (2015). [todo: ask for permission, or remake]

Transition-based parsing In many cases, AMR parses are structurally similar to syntactic dependency parses. Figure 13.6 shows one such example. This motivates an alternative approach to AMR parsing: simply modify the syntactic dependency parse until it looks like a good AMR parse. Wang et al. (2015) propose a transition-based method, based on incremental modifications to the syntactic dependency tree (you may review transition-based dependency parsing in § 11.3). At each step, the parser performs an action: for example, adding an AMR relation label to the current dependency edge, swapping the direction of a syntactic dependency edge, or cutting an edge and reattaching the orphaned subtree to a new parent. They train their system as a classifier, learning to choose the action as would be given by an **oracle** that is capable of reproducing the ground-truth parse. The 2016 SemEval evaluation compared a number of contemporary AMR parsing systems (May, 2016).

13.4 Applications of Predicate-Argument Semantics

Question answering **Factoid questions** have answers that are single words or phrases, such as *who discovered priors?*, *where was Barack Obama born?*, and *in what year did the Knicks last win the championship?* Shen and Lapata (2007) show that semantic role labeling can be used to answer such questions, by linking them to sentences in a corpus of text. They perform FrameNet semantic role labeling, making heavy use of dependency path features. For each sentence, they produce a weighted **bipartite graph**[15] between FrameNet seman-

[15]A bipartite graph is one in which the vertices can be divided into two disjoint sets, and every edge connect a vertex in one set to a vertex in the other.

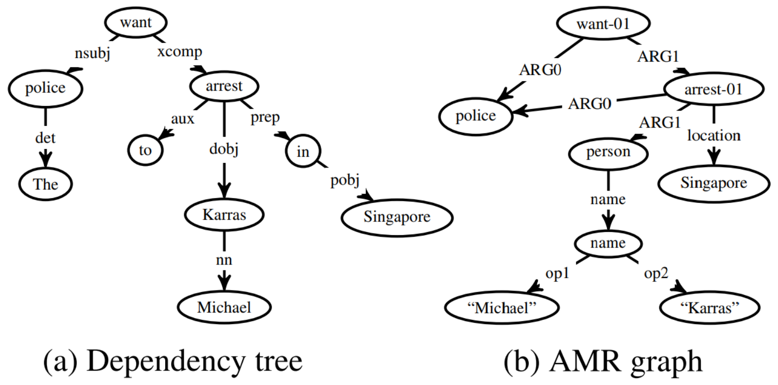


Figure 13.6: Syntactic dependency parse and AMR graph for the sentence *The police want to arrest Michael Karras in Singapore* (borrowed from Wang et al. (2015)) [todo: get permission]

tic roles and the words and phrases in the sentence. This is done by first scoring all pairs of semantic roles and assignments, as shown in the top half of Figure 13.8. They then find the bipartite edge cover, which is the minimum weighted subset of edges such that each vertex has at least one edge, as shown in the bottom half of Figure 13.8. After analyzing the question in this manner, Shen and Lapata then find semantically-compatible sentences in the corpus, by performing graph matching on the bipartite graphs for the question and candidate answer sentences. Finally, the **expected answer phrase** in the question — typically the *wh*-word — is linked to a phrase in the candidate answer source, and that phrase is returned as the answer.

Relation extraction The task of **relation extraction** involves identifying pairs of entities for which a given semantic relation holds. For example, we might like to find all $\langle i, j \rangle$ such that i is the INVENTOR-OF j. PropBank semantic role labeling can be applied to this task by identifying sentences whose verb signals the desired relation, and then extracting ARG1 and ARG2 as arguments. (To fully solve this task, these arguments must then be linked to entities, as described in chapter 17.) Christensen et al. (2010) compare the UIUC semantic role labeling system (which uses integer linear programming) against a simpler approach based on surface patterns (Banko et al., 2007). They find that the SRL system is considerably more accurate, but that it is several orders of magnitude slower. Conversely, Barnickel et al. (2009) apply SENNA, a convolutional neural network SRL system (Collobert and Weston, 2008) to the task of identifying biomedical relations (e.g., which genes inhibit or activate each other). In comparison with a strong baseline that applies a set of rules to syntactic dependency structures (Fundel et al., 2007), the SRL system is faster but less accurate. One possible explanation for these divergent results is that the Fundel et al. compare against a baseline which is carefully tuned for performance in a relatively narrow domain, while the system of Banko et al. is designed to analyze text across the entire

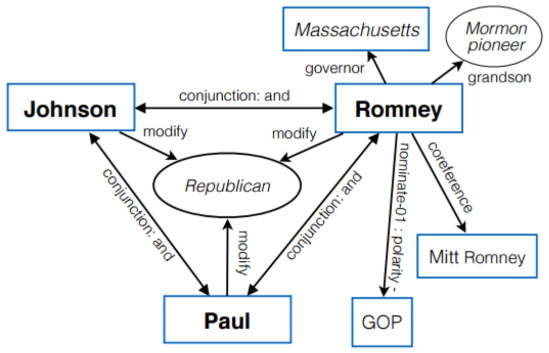


Figure 13.7: Fragment of AMR knowledge network for entity linking. Figure reprinted from Pan et al. (2015) [todo: permission]

web.

Entity linking Another core task in information extraction is to link mentions of entities (e.g., *Republican candidates like Romney, Paul, and* <u>*Johnson*</u> *...*) to entities in a knowledge base (e.g., `Lyndon Johnson` or `Gary Johnson`). This is often done by examining nearby "collaborator" mentions — in this case, *Romney* and *Paul*. By jointly linking all such mentions, it is possible to arrive at a good overall solution. Pan et al. (2015) apply AMR to this problem. For each entity, they construct a knowledge network based on its semantic relations with other mentions within the same sentence. They then rerank a set of candidate entities, based on the overlap between the entity's knowledge network and the semantic relations present in the sentence (Figure 13.7). When applied to manually labeled AMR annotations, this approach is superior to state-of-the-art supervised methods that have access to labeled examples of linked mentions. Pan et al. also show that the method performs well from automated AMR, and that an AMR-based approach far outperforms a similar method based on PropBank semantic role labeling.[todo: rework for clarity]

Exercises

1. Write out an event semantic representation for the following sentences. You may make up your own predicates.

(13.26) *Abigail shares with Max.*

(13.27) *Abigail reluctantly shares a toy with Max.*

(13.28) *Abigail hates to share with Max.*

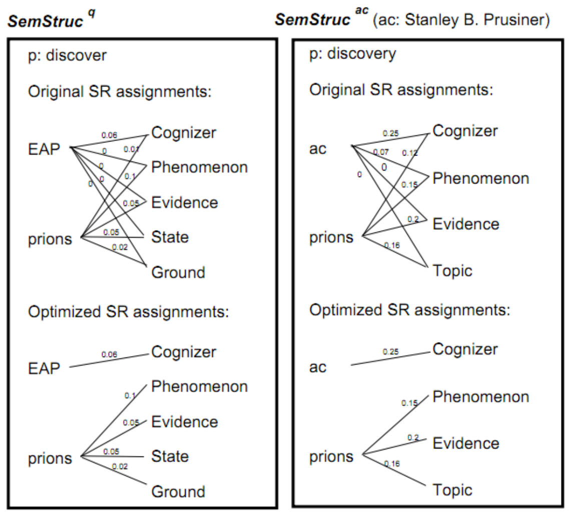


Figure 13.8: FrameNet semantic role labeling is used in factoid question answering, by aligning the semantic roles in the question (q) against those of sentences containing answer candidates (ac). "EAP" is the expected answer phrase, replacing the word *who* in the question. Figure reprinted from Shen and Lapata (2007) [todo: permission]

2. Find the PropBank framesets for *share* and *hate* at `http://verbs.colorado.edu/propbank/framesets-english-aliases/`, and rewrite your answers from the previous question, using the thematic roles ARG0, ARG1, and ARG2.

3. Compute the syntactic path features for Abigail and Max in each of the example sentences (13.26) and (13.28) in Question 1, with respect to the verb *share*. If you're not sure about the parse, you can try an online parser such as `http://nlp.stanford.edu:8080/parser/`.

4. Compute the dependency path features for Abigail and Max in each of the example sentences (13.26) and (13.28) in Question 1, with respect to the verb *share*. Again, if you're not sure about the parse, you can try an online parser such as `http://nlp.stanford.edu:8080/parser/`. As a hint, the dependency relation between *share* and *Max* is OBL according to the Universal Dependency treebank (version 2).

5. PropBank semantic role labeling includes **reference arguments**, such as,

   (13.29) [AM-LOC The bed] on [R-AM-LOC which] I slept broke.[16]

   The label R-AM-LOC indicates that word *which* is a reference to *The bed*, which expresses the location of the event. Reference arguments must have referents: the tag

[16]Example from 2013 NAACL tutorial slides by Shumin Wu

R-AM-LOC can appear only when AM-LOC also appears in the sentence. Show how to express this as a linear constraint, specifically for the tag R-AM-LOC. Be sure to correctly handle the case in which neither AM-LOC nor R-AM-LOC appear in the sentence.

6. Explain how to express the constraints on semantic role labeling in Equation 13.12 and Equation 13.13 in the general form $\mathbf{A}\boldsymbol{z} \geq \boldsymbol{b}$.

7. Download the FrameNet sample data (`https://framenet.icsi.berkeley.edu/fndrupal/fulltextIndex`), and train a bag-of-words classifier to predict the frame that is evoked by each verb in each example. Your classifier should build a bag-of-words from the sentence in which the frame-evoking lexical unit appears. [todo: Somehow limit to one or a few lexical units.] [todo: use NLTK if possible]

8. Download the PropBank sample data, using NLTK (`http://www.nltk.org/howto/propbank.html`). Use a deep learning toolkit such as PyTorch or DyNet to train an LSTM to predict tags. You will have to convert the downloaded instances to a BIO sequence labeling representation first.

9. Produce the AMR annotations for the following examples:

   (13.30) The girl likes the boy.

   (13.31) The girl was liked by the boy.

   (13.32) Abigail likes Maxwell Aristotle.

   (13.33) The spy likes the attractive boy.

   (13.34) The girl doesn't like the boy.

   (13.35) The girl likes her dog.

   For (13.32), recall that multi-token names are created using `op1`, `op2`, etc. You will need to consult Banarescu et al. (2013) for (13.34), and Schneider et al. (2015) for (13.35). You may assume that *her* refers to *the girl* in this example.

10. Using an off-the-shelf PropBank SRL system,[17] build a simplified question answering system in the style of Shen and Lapata (2007). Specifically, your system should do the following:

    - For each document in a collection, it should apply the semantic role labeler, and should store the output as a tuple.

[17] At the time of writing, the following systems are availabe: SENNA (`http://ronan.collobert.com/senna/`), Illinois Semantic Role Labeler (`https://cogcomp.cs.illinois.edu/page/software_view/SRL`), and mate-tools (`https://code.google.com/archive/p/mate-tools/`).

- For a question, your system should again apply the semantic role labeler. If any of the roles are filled by a *wh*-pronoun, you should mark that role as the expected answer phrase (EAP).
- To answer the question, search for a stored tuple which matches the question as well as possible (same predicate, no incompatible semantic roles, and as many matching roles as possible). Align the EAP against its role filler in the stored tuple, and return this as the answer.

To evaluate your system, download a set of three news articles on the same topic, and write down five factoid questions that should be answerable from the articles. See if your system can answer these questions correctly. (If this problem is assigned to an entire class, you can build a large-scale test set and compare various approaches.)

Chapter 14

Distributional and distributed semantics

A recurring theme in natural language processing is that the mapping from words to meaning is complex. In chapter 3, we saw that a single word form, like *bank*, can have multiple meanings. Conversely, a single meaning may be created by multiple surface forms, a lexical semantic relationship known as **synonymy**. Other lexical semantic relationships include **antonymy** (opposite meaning), **hyponymy** (instance-of), and **meronymy** (part-whole), each of which have semantic consequences for the interpretation of a sentence, utterance, or text.

Despite this complex mapping between words and meaning, the text analytic methods that we have considered thus far tend to consider words as the basic unit of analysis. All of the classifiers, sequence labelers, and parsers from the first chapters rely heavily on lexical features. The logical and frame semantic methods from chapter 12 and chapter 13 rely on hand-crafted lexicons that map from words to semantic predicates. But how can we analyze texts that contain words that we haven't seen before?

This chapter describes methods that create representations of word meaning by analyzing unlabeled data. The goal is for words with similar meanings to have similar representations; if this can be achieved, then it is possible for natural language processing systems to generalize from words that appear in costly resources such as labeled data or semantic lexicons to the broader vocabulary. The primary theory that makes it possible to achieve such representations from unlabeled data is the **distributional hypothesis**.

14.1 The distributional hypothesis

Here's a word you may not know: *tezgüino*.[1] Our position in deciding how to interpret sentences containing this word is similar to that of a natural language processing system

[1]The example is from Lin (1998).

	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	

Table 14.1: Distributional statistics for *tezgüino* and five related terms

when encountering a word that does not appear in the labeled training data.

Suppose we see that *tezgüino* is used in the following contexts:

- **C1**: *A bottle of* ________ *is on the table.*
- **C2**: *Everybody likes* ________.
- **C3**: *Don't have* ________ *before you drive.*
- **C4**: *We make* ________ *out of corn.*

What other words fit into these contexts? How about: *loud, motor oil, tortillas, choices, wine*? Each row of Table 14.1 is a vector that summarizes the contextual properties for each word, with a value of one for contexts in which the word can appear, and a value of zero for contexts in which it cannot. Based on these vectors, we can conclude:

- *wine* is very similar to *tezgüino*;
- *motor oil* and *tortillas* are fairly similar to *tezgüino*;
- *loud* is quite different.

These vectors, which we will call **word representations**, 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." This hypothesis has been stood the test of time: distributional statistics are a core part of language technology today, mainly because they make it possible to leverage large amounts of unlabeled data to learn about rare words that do not appear in labeled training data.

A striking demonstration of the power of distributional statistics is in their ability to represent lexical semantic relationships such as analogies. Figure 14.1 shows three examples. Distributional statistics are converted into vector **word embeddings**, using the GloVe algorithm, discussed later in this chapter (Pennington et al., 2014). These vectors

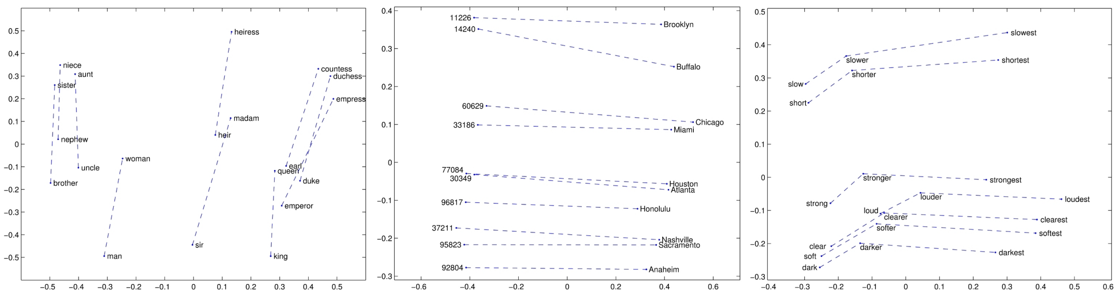


Figure 14.1: Lexical semantic relationships have regular linear structures in two dimensional projections of distributional statistics. From `http://nlp.stanford.edu/projects/glove/`.

are then projected into a two dimensional space. In each case, word-pair relationships correspond to regular linear patterns in this two dimensional space. No labeled data about the nature of these relationships was required to identify this underlying structure.

14.2 Design decisions for word representations

There are many approaches for computing word representations. To provide some structure to this space, it is useful to consider three dimensions along which word representations can be compared.

14.2.1 Representation

The most critical question is how words are to be represented. At present, the dominant choice is to represent words as k-dimensional vectors of real numbers, known as **word embeddings**. This representation dates back at least to the late 1980s (Deerwester et al., 1990), and is still used in currently popular techniques such as word2vec (Mikolov et al., 2013a). In word embeddings, similar words typically have high cosine similarity,

$$\cos(\boldsymbol{u}_i, \boldsymbol{u}_j) = \frac{\boldsymbol{u}_i \cdot \boldsymbol{u}_j}{||\boldsymbol{u}_i||||\boldsymbol{u}_j||}. \tag{14.1}$$

Dense vector word embeddings are well-suited for neural network architectures; they can also be applied in linear classifiers and structure prediction models (Turian et al., 2010), although some authors report that it can be difficult to learn linear models using real-valued features (Kummerfeld et al., 2015). A popular alternative is bit-string representations, such as **Brown clusters**, in which each word is represented by a variable-length sequence of zeros and ones (Brown et al., 1992). Another representational question is whether to estimate one embedding per word surface form or per word stem [todo: find cite], or

	The moment one learns ***English****, complications set in.*
Brown Clusters (Brown et al., 1992)	$\{$*learns*$\}$
WORD2VEC (Mikolov et al., 2013a) ($c = 2$)	$\{$*one, learns, complications, set*$\}$
Structured WORD2VEC (Ling et al., 2015a) ($c = 2$)	$\{(\textit{one}, -2), (\textit{learns}, -1), (\textit{complications}, +1), (\textit{set}, +2)\}$
Dependency contexts (Levy and Goldberg, 2014a)	$\{\textit{learns}/\textsc{dobj}^{-1}\}$

Table 14.2: Contexts for the word *English*, according to various word representations. For dependency context, $\textit{learns}/\textsc{dobj}^{-1}$ means that there is a relation of type DOBJ from the word *learns*.

whether to estimate distinct embeddings for each word sense (Huang et al., 2012a; Li and Jurafsky, 2015).

14.2.2 Context

The distributional hypothesis says that word meaning is related to the contexts in which the word appears, but the notion of context can be defined in a number of different ways. In the *tezgüino* example, contexts are entire sentences, but in practice there are far too many sentences for this to work — the resulting vectors would be too sparse. At the opposite end of the spectrum, the immediately preceding word could be taken as the context, and this is indeed the context considered in Brown clusters, which are discussed in **??**. WORD2VEC takes an intermediate approach, using local neighborhoods of words (e.g., $c = 5$) as contexts (Mikolov et al., 2013a). Contexts can also be much larger: for example, in **latent semantic analysis**, each word's context vector includes an entry per document, with a value of one if the word appears in the document (Deerwester et al., 1990); in **explicit semantic analysis**, these documents are Wikipedia pages (Gabrilovich and Markovitch, 2007).

Words in context can be labeled by their position with respect to the target word w_m (e.g., two words before, one word after), and this appears to make the resulting word representations more sensitive to syntactic differences (Ling et al., 2015a). Another way to incorporate syntax is to perform parsing as a preprocessing step, and then form context vectors from the set of dependency edges (Levy and Goldberg, 2014a) or predicate-argument relations (Lin, 1998). The resulting context vectors for several of these methods are shown in **??**.

The choice of context has a profound effect on the resulting representations, which can be viewed in terms of word similarity. Applying latent semantic analysis (**??**) to contexts of length two and length 30, one obtains the following nearest-neighbors for the word *dog*:[2]

- $(c = 2)$: *cat, horse, fox, pet, rabbit, pig, animal, mongrel, sheep, pigeon*
- $(c = 30)$: *kennel, puppy, pet, bitch, terrier, rottweiler, canine, cat, to bark, Alsatian*

Which word list is better? Every item on the $c = 2$ is an animal, while the $c = 30$ list starts with *kennel* and even includes the verb *to bark*.

14.2.3 Estimation procedure

14.3 Distributional semantics

14.3.1 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.[3]

In Brown clustering, the context is just the immediately adjacent words. The similarity metric is built on a generative probability model:

- Assume each word w has a class $C(w)$
- Assume a generative model $\log \mathrm{p}(w) = \sum_i \log \mathrm{p}(w_i \mid c_i) + \log \mathrm{p}(c_i \mid c_{i-1})$
  (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,

$$\mathrm{p}(w_i \mid c_i) = \begin{cases} \frac{\text{count}(w_i)}{\text{count}(c_i)}, & c_i = C(w_i) \\ 0, & \text{otherwise.} \end{cases} \tag{14.2}$$

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

[2]The example is from lecture slides by Marco Baroni, Alessandro Lenci, and Stefan Evert, who applied latent semantic analysis to the British National Corpus. You can play with an online demo here: `http://clic.cimec.unitn.it/infomap-query/`

[3]You can download Brown clusters at `http://metaoptimize.com/projects/wordreprs/`.

Algorithm 12 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
```

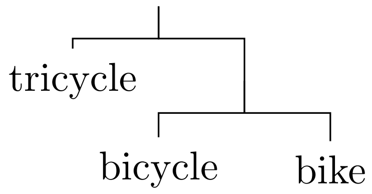


Figure 14.2: A small subtree produced by bottom-up Brown clustering

will not apply the expectation maximization algorithm which was used in unsupervised hidden markov model learning (§ 6.6). Instead, Brown et al. (1992) use a hierarchical clustering algorithm, shown in Algorithm 12. 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 14.2.

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 14.2 would be 0 for *tricycle*, 10 for *bicycle*, and 11 for *bike*. Subtrees from Brown clustering on a larger dataset are shown in Figure 14.3. The examples are drawn from a paper by Miller et al. (2004), who use Brown cluster bitstring prefixes as features for named entity recognition; 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 12 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.

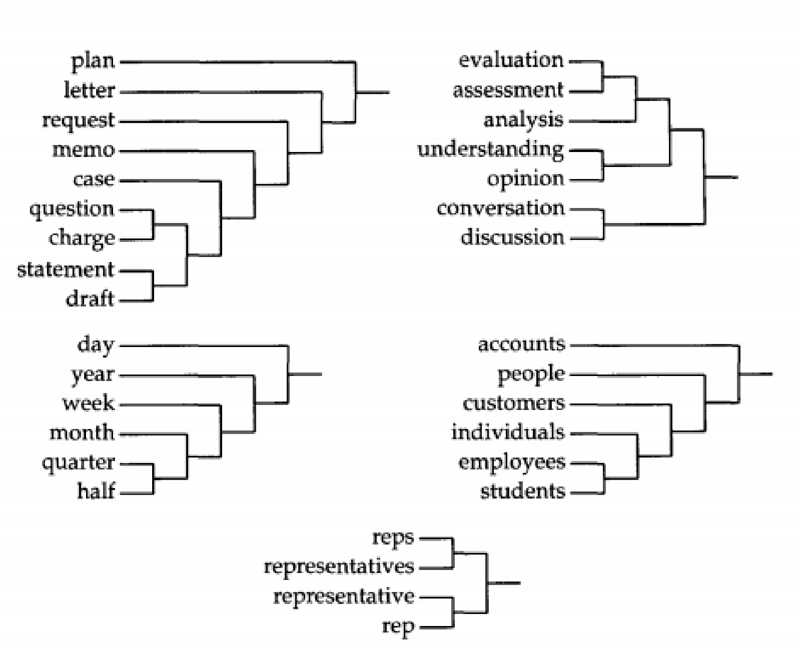


Figure 14.3: Brown subtrees from Miller et al. (2004)

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,

$$H(X) = -E[\log P(X)] = -\sum_{x} P(X = x) \log P(X = x). \tag{14.3}$$

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,

$$I(X;Y) = \sum_{y \in Y}\sum_{x \in X} \mathrm{p}_{X,Y}(x,y) \log \left(\frac{\mathrm{p}_{X,Y}(x,y)}{\mathrm{p}_X(x)\mathrm{p}_Y(y)}\right). \tag{14.4}$$

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

Algorithm 13 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 $c'$
```

By $I(C)$, we are using a shorthand for the mutual information of adjacent word classes, $\langle C_{m-1}, C_m \rangle$,

$$I(C) = \sum_{C_m = c, C_{m-1} = c'} P(C_m = c, C_{m-1} = c') \log \left(\frac{P(C_m = c, C_{m-1} = c')}{P(C_m = c) P(C_{m-1} = c')} \right) \tag{14.5}$$

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 13. 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 $\mathcal{O}(V \log V + VK^2)$.

14.3.2 Syntactic distributional statistics

Local context is contingent on syntactic decisions that may have little to do with semantics:

(14.1) *I gave Tim the ball.*

(14.2) *I gave the ball to Tim.*

(You may recall from **??** 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:

$$\hat{p}(n, v) = \sum_{c \in \mathcal{C}} p(v \mid c) \times p(c, n) \tag{14.6}$$

$$= \sum_{c \in \mathcal{C}} p(v \mid c) \times p(n \mid c) \times p(c), \tag{14.7}$$

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:

$$p(i, j \mid r) = \frac{n(i, j, r)}{\sum_{i', j'} n(i', j', r)} \tag{14.8}$$

$$p(i \mid r) = \sum_{j} p(i, j \mid r) \tag{14.9}$$

Now, let $T(i)$ be the set of pairs $\langle j, r \rangle$ such that $p(i, j \mid r) > p(i \mid r) \times p(j \mid r)$: 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 $\frac{|T(u) \cap T(v)|}{|T(u)| + |T(v)| - |T(u) \cap T(v)|}$

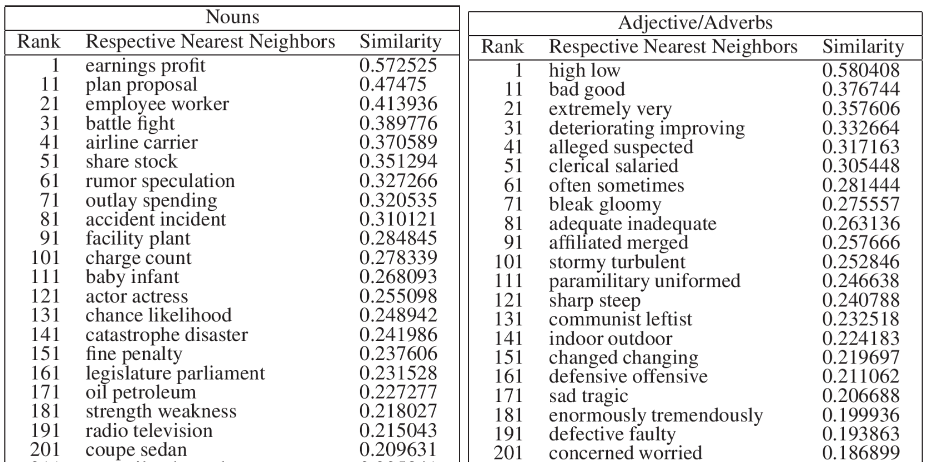
However, Lin's chosen metric is more complex than any of these well-known alternatives:

$$\frac{\sum_{\langle r, w \rangle \in T(u) \cup T(v)} I(u, r, w) + I(v, r, w)}{\sum_{\langle r, w \rangle \in T(u)} I(u, r, w) + \sum_{\langle r, w \rangle \in T(v)} I(v, r, w)}, \tag{14.10}$$

where $I(u, r, w)$ is the mutual information between u and w, conditioned on r.

Results of the algorithm are shown in Figure 14.4. 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!



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 14.4: Similar word pairs from the clustering method of Lin (1998)

14.4 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).

14.4.1 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

$$\mathbf{U}_K \in \mathbb{R}^{V \times K}, \qquad \mathbf{U}_K \mathbf{U}_K^\top = \mathbb{I} \tag{14.11}$$

$$\mathbf{S}_K \in \mathbb{R}^{K \times K}, \qquad \mathbf{S}_K \text{ is diagonal, non-negative} \tag{14.12}$$

$$\mathbf{V}_K \in \mathbb{R}^{D \times K}, \qquad \mathbf{V}_K \mathbf{V}_K^\top = \mathbb{I} \tag{14.13}$$

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

$$\mathbf{U}_K, \mathbf{S}_K, \mathbf{V}_K = \underset{\mathbf{U}_k, \mathbf{S}_K, \mathbf{V}_K}{\operatorname{argmin}} \; ||\mathbf{X} - \mathbf{U}_K \mathbf{S}_K \mathbf{V}_K^\top||_F, \tag{14.14}$$

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 $\min\left(\mathcal{O}(D^2V), \mathcal{O}(V^2N)\right)$. 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.

14.4.2 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

$$u_\theta(w, c) = \exp\left(\boldsymbol{a}_w^\top \boldsymbol{b}_c\right) \tag{14.15}$$

$$\tag{14.16}$$

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[4] 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,

[4]https://code.google.com/p/word2vec/

$$J = \frac{1}{M} \sum_m \sum_{-c \leq j \leq c, j \neq 0} \log \mathrm{p}(w_{m+j} \mid w_m) \tag{14.17}$$

$$\mathrm{p}(w_{m+j} \mid w_m) = \frac{u_\theta(w_{m+j}, w_m)}{\sum_{w'} u_\theta(w', w_m)} \tag{14.18}$$

$$= \frac{u_\theta(w_{m+j}, w_m)}{Z(w_m)} \tag{14.19}$$

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.

$$J = \frac{1}{M} \sum_m \log \mathrm{p}(w_m \mid c) \tag{14.20}$$

$$= \frac{1}{M} \sum_m \log u_\theta(w_m, c) - \log Z(c) \tag{14.21}$$

$$u_\theta(w_m, c) = \exp\left(\sum_{-c \leq j \leq c, j \neq 0} \boldsymbol{a}_{w_m}^\top \boldsymbol{b}_{w_{m+j}} \right) \tag{14.22}$$

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 (§ 5.3) 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 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,

$$P(W_{m+1} = i \mid \boldsymbol{c}_m) = \frac{\exp\left(\boldsymbol{c}_m^\top \boldsymbol{v}_i\right)}{\sum_{i'} \exp\left(\boldsymbol{c}_m^\top \boldsymbol{v}_{i'}\right)}, \tag{14.23}$$

where $\boldsymbol{v}_i$ is the "output embedding" of word i.

14.4.3 *Estimating word embeddings

[todo: link to rnnlm, show pictures] 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

$$D = \begin{cases} 1, & w \text{ is drawn from the empirical distribution } \hat{\mathrm{p}}(w \mid c) \\ 0, & w \text{ is drawn from the noise distribution } q(w) \end{cases} \tag{14.24}$$

Now we will optimize the objective

$$\sum_{(w,c)\in\mathcal{D}} \log P(D = 1 \mid c, w) + \sum_{i=1, w' \sim q}^{k} \log P(D = 0, \mid c, w'), \tag{14.25}$$

setting

$$P(D = 1 \mid c, w) = \frac{u_\theta(w, c)}{u_\theta(w, c) + k \times q(w)} \tag{14.26}$$

$$P(D = 0 \mid c, w) = 1 - P(D = 1 \mid c, w) \tag{14.27}$$

$$= \frac{k \times q(w)}{u_\theta(w, c) + k \times q(w)}, \tag{14.28}$$

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

$$P(D = 1 \mid c, w) = \frac{u_\theta(w, c)}{u_\theta(w, c) + 1} \tag{14.29}$$

$$P(D = 0 \mid c, w) = \frac{1}{u_\theta(w, c) + 1} \tag{14.30}$$

The derivative with respect to a is obtained from the objective

$$L = \sum_m \log \mathrm{p}(D = 1 \mid c_m, w_m) + \log \mathrm{p}(D = 0 \mid c, w') \tag{14.31}$$

$$= \sum_m \log u_\theta(w_m, c_m) - \log(1 + u_\theta(w_m, c_m)) - \log(1 + u_\theta(w', c_m)) \tag{14.32}$$

$$\frac{\partial L}{\partial \boldsymbol{a}_i} = \sum_{m:w_m=i} \boldsymbol{b}_{c_m} - \frac{1}{1 + u_\theta(w_m, c_m)} \frac{\partial u_\theta(i, c_m)}{\partial \boldsymbol{a}_i} + \sum_m \frac{q(i)}{1 + u_\theta(i, c_m)} \frac{\partial u_\theta(i, c_m)}{\partial \boldsymbol{a}_i} \tag{14.33}$$

$$= \sum_{m:w_m=i} \boldsymbol{b}_{c_m} - P(D = 1 \mid w_m = i, c_m)\boldsymbol{b}_{c_m} - \sum_m q(i) P(D = 0 \mid i, c_m)\boldsymbol{b}_{c_m} \tag{14.34}$$

$$= \sum_m (\delta(w_m = i) - q(i)) P(D = 0 \mid w_m = i, c_m)\boldsymbol{b}_{c_m}. \tag{14.35}$$

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.

14.4.4 *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, 2014b). For example, Levy and Goldberg (2014b) show that skip-gram with negative sampling is equivalent to factoring a matrix X, where

$$X_{i,j} = PMI(W = i, C = j) - \log k, \tag{14.36}$$

where k is a constant offset equal to the number of negative samples drawn in Equation 14.25, and PMI is the **pointwise mutual information** of the events of the word $W = i$ and the context $C = j$,

$$PMI(W = i, C = j) = \log \frac{P(W = i, C = j)}{P(W = i)P(C = j)} \tag{14.37}$$

$$= \log \frac{n(W = i, C = j)}{M} \frac{M}{n(W = i)} \frac{M}{n(C = j)} \tag{14.38}$$

$$= \log \frac{n(W = i, C = j)}{n(W = i)} \frac{M}{n(C = j)}. \tag{14.39}$$

Word embeddings can be obtained by solving the truncated singular value decomposition $\mathbf{U\Sigma V}^\top = \mathbf{X}$, 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 (2014b) 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 15

Reference Resolution

References are one of the most noticeable forms of linguistic ambiguity, afflicting not just automated natural language processing systems, but also fluent human readers. For this reason, warnings to avoid "ambiguous pronouns" are ubiquitous in manuals and tutorials on writing style. But referential ambiguity is not limited to pronouns, as shown in the following text:

(15.1) *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]***$_3$*...* ***[Cook]***$_4$ *is on* ***[his]***$_5$ *first trip to* ***[the country]***$_6$ *since taking over...*[1]

Each of the bolded substrings in the passage refers to an entity that is introduced earlier in the story. These references include the pronouns *he* and *his*, but also the shortened name *Cook*, and most challengingly, **nominals** such as *the firm* and *the firm's biggest growth market*. Only by resolving several of these references can we reach the (correct) inference that China is Apple's biggest growth market.

The task of reference resolution is often broken into two components:

- **Coreference resolution**, which is the task of linking spans of text such as the *the firm* to other spans, such as *Apple Inc*. A subset of coreference resolution is the task of **anaphora resolution**, which typically involves resolving only pronoun anaphora such as *he* and *her*.
- **Entity linking**,[2] which is the task of linking spans of text to entities in a knowledge base. This step is a prerequisite for the model-based semantic parsing that was considered in chapter 12.

[1]`http://www.reuters.com/article/us-apple-china-idUSBRE82Q06420120327`, retrieved on March 26, 2017

[2]Amusingly, there are many names for this task: deduplication, approximate string match, entity clustering, record linking, multidocument coreference resolution, etc.

These tasks have traditionally been distinguished because they seem to require different sorts of knowledge to perform, and different resources to evaluate. As we will see, coreference resolution — especially anaphora resolution — is constrained by syntax and by compatibility of attributes such as gender, number, and animacy. Coreference resolution can be evaluated by comparing against a ground truth that is specified at the document level, without reference to any external information. In contrast, solving entity linking requires making inferences about name compatibility and about semantic properties of each entity — although these inferences are sometimes necessary for coreference resolution too. Evaluation of entity linking requires linking textual references to some predefined external ontology. Of the two tasks, research on coreference resolution is more mature, and will therefore be the focus of this chapter. Approaches to entity linking and related tasks are summarized in **??**.

15.1 Forms of referring expressions

The three main forms of referring expressions — pronouns, names, and nominals — each pose unique challenges for the reader. As a coarse-grained summary, pronouns are constrained by syntax and semantic attributes; name references constrained by rules for matching; nominals are linked by world knowledge.

15.1.1 Pronouns

Pronouns are a closed class of words that are used for references. A natural way to think about pronoun resolution is what Kehler (2007) calls the SMASH approach:

- **S**earch for candidate referents;
- **M**atch against hard agreement constraints;
- **A**nd **S**elect using **H**euristics, which are "soft" constraints such as recency and parelelism.

In the search step, candidates are identified from the preceding text or speech.[3] In models such as **centering theory**, any entity that has previously been **evoked** can be **accessed** in any subsequent unit of text (Grosz et al., 1995). However, cognitive constraints may imply that entities which have not been mentioned recently are unlikely to be accessed without be re-introduced, and correspondingly, computational constraints may encourage algorithms to consider only referents that are relatively recent.

[3]Pronouns whose referents come later are known as **cataphora**, e.g.,

(15.1) *Many years later, as **he** faced the firing squad, **Colonel Aureliano Buendía** was to remember that distant afternoon when his father took him to discover ice.*

(This is the first sentence of *One Hundred Years of Solitude*, by Gabriel García Márquez.)

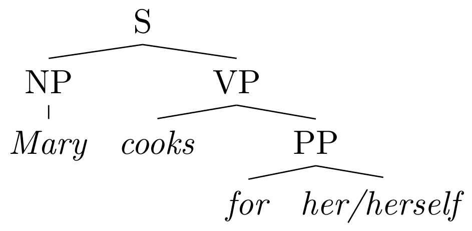


Figure 15.1: *Mary* c-commands *her/herself*

Matching constraints for pronouns

Semantic constraints include morphologically marked information such as number, person, gender, and animacy.

(15.2) 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*.

Other features include person, gender, and animacy, as in the following examples:

(15.3) *Sally met my brother. He charmed her.*

(15.4) *Sally met my brother. She charmed him.*

(15.5) *We$_1$ *told* them$_1$ *not to go*.

(15.6) *Putin brought a bottle of vodka. It was from Russia.*

Another source of constraints comes from syntax. To understand these constraints, it is helpful 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 15.1. 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*,

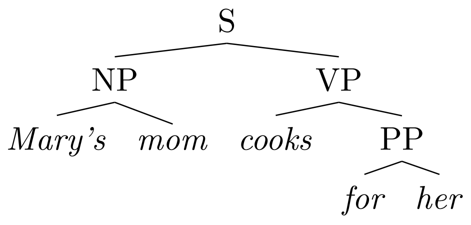


Figure 15.2: *Mary* does not c-command *her*, but *Mary's mom* does.

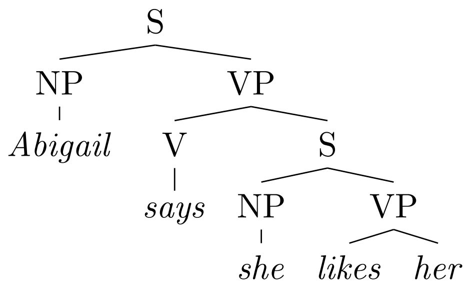


Figure 15.3: The scope of *Abigail* is limited by the S non-terminal. Either *she* or *her* (but not both) can bind to *Abigail*.

because non-reflexive pronouns cannot refer to antecedents that c-command them. On the other hand, the reflexive *herself* **must** refer to *Mary*.

Now consider the example, shown in Figure 15.2. Here, *Mary* does **not** c-command *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. However, *her* does not have to refer to *Mary*.

A more complex example is shown in Figure 15.3. This indicates how the constraints defined here have a limited domain. [todo: explain how this is limited] 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.

Heuristics

After applying constraints, there will be a number of candidate referents for each pronoun. In the SMASH paradigm, heuristics are then applied to compare among these possibilities.

Recency is a particularly strong heuristic. All things equal, readers will prefer the more recent referent for a given pronoun, particularly when comparing referents that occur in different sentences. Jurafsky and Martin (2009) offer the following example:

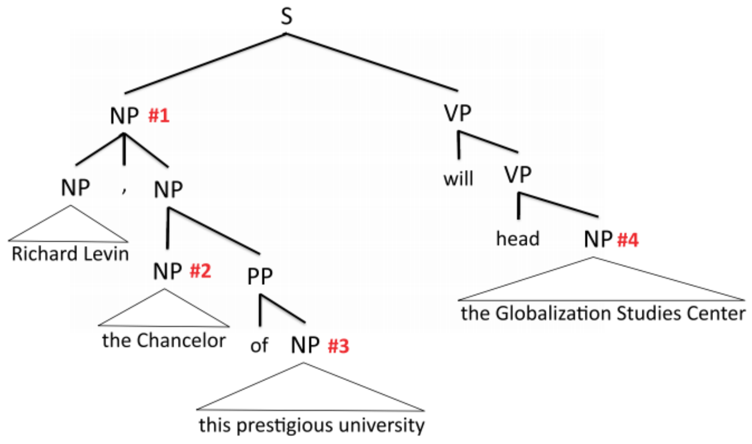


Figure 15.4: Left-to-right breadth-first tree traversal, proposed by Hobbs (1978), as implemented by Lee et al. (2013)

(15.7) *The doctor found an old map in the captain's chest. Jim found an even older map hidden on the shelf.* ***It*** *described an island.*

Readers are expected to prefer the second, older map as the referent for the pronoun *it*.

However, subjects are often preferred over objects, and this can contradict the preference for recency when two candidate referents are in the same sentence. For example,

(15.8) *Asha loaned Mei a book on Spanish.* ***She*** *is always trying to help people.*

Here, we may prefer to link *she* to *Asha* rather than *Mei*, because of *Asha*'s position in the subject role of the preceding sentence. (Arguably, this preference would be reversed if the second sentence were *She is visiting Argentina next month*.)

A third heuristic is parallelism:

(15.9) *Asha loaned Mei a book on Spanish. Olya loaned* ***her*** *a book on Portuguese.*

Here *Mei* is preferred as the referent for *her*, contradicting the preference for the subject *Asha* in the preceding sentence.

Hobbs (1978) unifies recency and subject-role heuristics by traversing the document in a syntax-driven fashion: each preceding sentence is traversed breadth-first, left-to-right (Figure 15.4). In this way, *Asha* would be preferred as the referent for *she* in (15.8), while the older map would be preferred as the referent for *it* in *ex:coref-recency*. **Centering theory** offers an alternative unification of recency and syntactic prominence, maintaining ordered lists of candidates referents throughout the text or discourse (Grosz et al., 1995). Centering can also be viewed as a generative model, in that it predicts the form of the referring expression that will be used for each entity reference in a sentence.

In early work on reference resolution, Lappin and Leass (1994) set weights for a half-dozen syntactic preferences by hand, choosing the referent with the highest overall weight. More recent work uses machine learning approaches to quantify the importance of each of these factors, as discussed in **??**. However, pronoun resolution often cannot be performed successfully using syntactic heuristics alone. This is shown by the classic example pair:

(15.10) *The **city council** denied the protesters a permit because **they** feared violence.*

(15.11) *The city council denied **the protesters** a permit because **they** advocated violence.*[4]

Non-referential pronouns

While pronouns are generally used to refer to things, they need not refer to entities, as shown in the following examples:

(15.12) *They told me that I was too ugly, but I didn't believe **it**.*

(15.13) *Alice saw Bob get angry, and I saw **it** too.*

(15.14) *They told me that I was too ugly, but **that** was a lie.*

(15.15) *Jess said she worked in security.*
*I suppose **that**'s one way to put it.*

Pronouns may also have **generic** referents, meaning that they do not refer to entities in any model, but rather, to possible entities:

(15.16) *A good father takes care of **his** kids.*

(15.17) *On the moon, **you** have to carry **your** own oxygen.*

(15.18) *Every farmer who owns a donkey beats it.* (Geach, 1962)

Finally, pronouns need not refer to anything at all:

(15.19) ***It**'s raining.*

(15.20) ***It**'s crazy out there.*

(15.21) ***It**'s money that she's really after.*

(15.22) ***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:

(15.23) *You can make it in advance.*

(15.24) *You can make it in showbiz.*

[4]This pair is attributed to Winograd (1972), but I downloaded that article and didn't find it.

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:

(15.25) *You can make them in advance.*

(15.26) ?*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 comparing distributional statistics for 5-grams around the word *it*, testing how often other pronouns or nouns ever appear in the same position as *it*. In cases where other pronouns are frequent, the *it* is likely referential.

15.1.2 Proper Names

If a proper name is used as a referring expression, it often refers to another proper name, so that the coreference problem is simply to determine whether the two names match. Subsequent proper name references often use a shortened form, as in the running example:

(15.27) *Apple Inc Chief Executive* ***Tim Cook*** *has jetted into China ...* ***Cook*** *is on his first business trip to the country since taking over ...*

In this news article, the family name *Cook* is used as a referring expression; in informal conversation, it might be more typical to use the given name *Tim*, while more formal venues, such as *The Economist*, would use the title form *Mr Cook*.

Thus, exact match is unlikely to identify many proper name references. A typical solution is to match the syntactic **head words** of the reference with the referent. Recall that the head word of a phrase can be identified by applying head percolation rules to the phrasal parse tree (chapter 10); alternatively, the head can be identified as the root of the dependency subtree covering the name. For sequences of proper nouns, the head word will be the final word, which in the example is *Cook*.

While useful, there are a number of caveats to the practice of matching head words of proper names.

- In the European tradition, family names tend to be more specific than given names, and family names usually come last. However, other traditions have other practices: for example, in Chinese names, the family name typically comes first; in Japanese, honorifics come after the name, as in *Nobu-San* (*Mr. Nobu*).
- In many organization names, it is also the case that the head word is not the most informative: for example, *Georgia Tech* and *Virginia Tech* are distinguished by the modifiers *Virginia* and *Georgia*, and not the heads. This concern applies even when the referring expression is a substring of the candidate referent: *Lebanon* does not refer to the same entity as *Southern Lebanon*, and Lee et al. (2011) add a rule to deal with the specific case of geographical modifiers.

- Finally, proper names can be nested, as in *[the CEO of [Microsoft]]*. Haghighi and Klein (2009) introduce a constraint to prevent nested noun phrases from being marked as coreferential.

Despite these difficulties, proper names are the easiest category of references to resolve (Stoyanov et al., 2009). In machine learning systems, one solution is to include a range of matching features, including exact match, head match, string inclusion, and even matching on "bags" of tokens, so that, e.g., *Tim Cook* matches *Cook, Tim* (Bontcheva et al., 2002). In addition to matching features, competitive systems include large lists, or **gazetteers**, of acronyms (e.g, *the National Basketball Association/NBA*), demonyms (e.g., *the Israelis/Israel*), and other aliases (e.g., *the Georgia Institute of Technology/Georgia Tech*).[5] The learning algorithm can then determine the appropriate weights for each matching feature.

15.1.3 Nominals

In coreference resolution, noun phrases that are neither pronouns nor names are referred to as **nominals**. In the running example, nominal references include:

- *the firm* (*Apple Inc*)
- *the firm's biggest growth market* (*China*)
- *the country* (*China*).

Nominals are generally more difficult to resolve than pronouns and names (Durrett and Klein, 2013, e.g.,), and the examples above suggest why this may be the case: world knowledge is required to identify *Apple Inc* as a *firm*, and *China* as a *growth market*. Other difficult examples include the use of colloquial expressions, such as coreference between *Clinton transition officials* and *the Clinton camp* (Soon et al., 2001). But there are also cases that can be handled by surface features such as head word match: for example, *the tax cut bill* may be referenced later by *the Republican bill* or even *the bill*.

Attempts to use semantics to improve nominal coreference have met with limited success. Durrett and Klein (2013) employ WordNet synonymy and hypernymy relations on head words, named entity types (e.g., person, organization), and unsupervised clustering over nominal heads. These features give only limited improvement over simple baseline using surface features such as string match.

15.2 Learning for coreference resolution

Coreference resolution is a non-traditional learning problem, because it is not obvious what consitutes an "instance." A number of proposals have been put forward. In dis-

[5]Lists of aliases were used heavily in the Message Understanding Conference (MUC) systems of the 1990s, which helped to define the coreference resolution task (Grishman and Sundheim, 1996). They are still used in some of the most competitive systems at the time of this writing (e.g. Martschat and Strube, 2015).

cussing these approaches, references and candidate referents are called **mentions**; chains of references are called **entities**.[6] [todo: add figure] Ground truth annotations identify the entities. In our running example, this would be:

- *Apple Inc Chief Executive Tim Cook*, *he*, *Cook*
- *Apple Inc*, *the firm*
- *China*, *the firm's biggest growth market*, *the country*.

"Singleton" mentions (e.g., *government officials*) are annotated in the ACE Corpus (**?**), but not in the OntoNotes corpus (Hovy et al., 2006).

Coreference resolution can be viewed as a structure prediction problem, where the goal is to identify a set of coreference chains $\boldsymbol{c}$ among all possible coreference structures $\mathcal{C}(\boldsymbol{w})$,

$$\underset{\boldsymbol{c}\in\mathcal{C}(\boldsymbol{w})}{\operatorname{argmax}}\, \boldsymbol{\theta}\cdot \boldsymbol{f}(\boldsymbol{c},\boldsymbol{w}). \tag{15.1}$$

Each chain c_i consists of a set of mentions $\{m_j\}$. Typically it is the coreference resolution system's job to identify the mentions from unannotated text, although systems are sometimes evaluated with "gold" mentions from the annotators.

The main approaches to coreference resolution can be distinguished by how they decompose the feature function $\boldsymbol{f}(\boldsymbol{c},\boldsymbol{w})$. In mention-based models, features are defined over pairs of entities. This can facilitate inference, but mention-based models can suffer from incoherent entity chains, such as $\{$*Hillary Clinton* $\leftarrow$ *Clinton* $\leftarrow$ *Mr Clinton*$\}$. In entity-based models, the goal is to ensure that the entire entity chain is coherent. This can make inference more difficult, since the number of possible entity groupings is exponential in the number of mentions. A second distinction is whether the training instances are pairs (mention-mention pairs or mention-entity pairs), or whether learning is performed by ranking all possible candidates (mentions or entities) for a given mention.

15.2.1 Mention-pair and mention ranking models

In the **mention-pair model**, a label $y_{ij} \in \{0, 1\}$ is assigned to each pair of mentions $\langle i, j\rangle, i < j$. If i and j corefer, then $y_{ij} = 1$, and we say that i is the **antecedent** of j; otherwise, $y_{ij} = 0$. Thus, the mention-pair model reduces coreference resolution to binary classification, enabling the application of off-the-shelf machine learning algorithms: Soon et al. (2001) use decision trees, and Bengtson and Roth (2008) use the averaged perceptron.

Under the constraint that each mention has at most one antecedent, the **antecedent structure** $\{y_{ij}\}$ induces a unique set of entities $\boldsymbol{c}$. However, the converse is not true: a

[6] In many annotations, the term **markable** is used to refer to spans of text that can **potentially** mention an entity. The set of markables includes non-referential pronouns such as pleonastic *it*, which does not mention any entity. Part of the job of the coreference system is to avoid incorrectly linking these non-referential markables to any mention chains.

single set of entities $\boldsymbol{c}$ may be compatible with multiple antecedent structures. Since the ground truth annotations give $\boldsymbol{c}$ but not $\boldsymbol{y}$, additional heuristics must be employed to convert the labeled data into training examples. Furthermore, we must impose the constraint that each mention have at most one antecedent. One solution is to pair the classifier with a search heuristic, based on SMASH: search backwards from j until finding an antecedent i which corefers with j with high confidence, and then stop searching. During training, for each reference j with antecedent i, we include negative examples $y_{i'j} = 0$ for all $i < i' < j$.

In **mention ranking**, the classifier learns to identify a single antecedent $a_i \in \{1, 2, \ldots, i-1, i\}$ for each referring expression i, where $a_i = i$ indicates that the mention i does not refer to any previously-introduced entity (Denis and Baldridge, 2007). Specifically, the model chooses,

$$\hat{a}_i = \underset{a \in \{1,2,\ldots,i\}}{\operatorname{argmax}} \; \boldsymbol{\theta} \cdot \boldsymbol{f}(i, a, \boldsymbol{w}), \tag{15.2}$$

where $\boldsymbol{f}(i, a, \boldsymbol{w})$ defines a set of features on the mention pair $\langle i, a \rangle$. A special set of features can be employed for the case $a_i = i$, although later work on mention ranking has employed a two-stage model, in which an "anaphoricity" classifier determines whether the mention i refers to a previously defined entity; if so, then the ranking decision is performed over the set $1, 2, \ldots, i-1$ (Denis and Baldridge, 2008).

As with the binary coreference variables $\{y_{ij}\}$, the antecedent variables $\{a_i\}$ relate to the entity chains in a many-to-one mapping: each set of assignment variables induces a single entity clustering, but an entity clustering can correspond to many different settings of assignment variables. When mention i has multiple possible antecedents in the clustering $\boldsymbol{c}$, a typical approach is to select the most recent compatible antecedent. However, by using a probabilistic ranking model, $\mathrm{p}(\{a_i\} \mid \boldsymbol{w})$, Durrett and Klein (2013) are able to sum over the set of all antecedent structures $\mathcal{A}(\boldsymbol{c})$ that are compatible with the gold coreference clustering $\boldsymbol{c}$,

$$\mathrm{p}(a_i \mid i, \boldsymbol{w}) = \frac{\exp\left(\boldsymbol{\theta} \cdot \boldsymbol{f}(i, a_i, \boldsymbol{w})\right)}{\sum_{a' \leq i} \exp\left(\boldsymbol{\theta} \cdot \boldsymbol{f}(i, a', \boldsymbol{w})\right)} \tag{15.3}$$

$$\mathrm{p}(\boldsymbol{a} \mid \boldsymbol{w}) = \prod_{i}^{M} \mathrm{p}(a_i \mid i, \boldsymbol{w}) \tag{15.4}$$

$$\mathrm{p}(\boldsymbol{c} \mid \boldsymbol{w}) = \sum_{\boldsymbol{a} \in \mathcal{A}(\boldsymbol{c})} \mathrm{p}(\boldsymbol{a} \mid \boldsymbol{w}). \tag{15.5}$$

In this way, Durrett and Klein learn a model that tries to assign high scores for all valid antecedent structures.

15.2.2 Entity-based models

Many of the practical difficulties with mention-based models stem from the fact that they treat coreference resolution like a classification or ranking problem, when in fact it is a clustering problem: the goal is to group the mentions together into clusters that correspond to the underlying entities. Entity-based approaches attempt to identify these clusters directly. Such methods require defining features at the entity level, measuring whether the set of mentions are internally consistent. Cardie and Wagstaff (1999) provide an early example of entity-based coreference, incrementally merging clusters of mentions under the constraint that all pairs of mentions in the entity are compatible in number, gender, animacy, etc. They define a set of soft preferences for merging when there are multiple clusters that are compatible. More recent methods for entity-based coreference resolution have applied machine learning in the context of incremental search over the space of coreference clusterings (e.g., Clark and Manning, 2015).

The gap between entity-based and mention-pair models can be partially bridged by enforcing transitivy on the mention-pair variables: if $y_{ij} = 1$ and $y_{jk} = 1$, then $y_{ik} = 1$. This constraint can be written as a linear inequality,

$$\forall i < j < k, y_{ik} \geq y_{ij} + y_{jk} - 1. \tag{15.6}$$

The transitivity constraint ensures that each mention is linked to all antecedents in its cluster. We can then formulate the inference problem as,

$$\max \sum_{i,j} \boldsymbol{\theta} \cdot \boldsymbol{f}(y_{ij}, \boldsymbol{w}) \tag{15.7}$$

$$s.t. \forall i < j < k, y_{ik} \geq y_{ij} + y_{jk} - 1. \tag{15.8}$$

In this formulation, features are still defined over mention pairs — rather than over entire entities — but transitivity ensures that all pairs in the cluster are compatible, avoiding incoherent clusters like {*Hillary Clinton* ← *Clinton* ← *Mr Clinton*}. However, this coherence comes at a computational price: the constrainted optimization problem is NP-hard. **Integer linear programming** (ILP), which we saw in chapter 13, is one solution (Klenner, 2007; Finkel and Manning, 2008); correlational clustering is another (McCallum and Wellner, 2004).

[todo: discuss recent methods for using deep learning to acquire entity representations (**??**)]

15.2.3 Deterministic methods

Unlike many other areas of natural language processing, it is possible to build competitive systems for coreference resolution without machine learning (Haghighi and Klein, 2009). One such architecture is shown in Figure 15.5. The basic idea is to apply a series of rule-based methods, or "sieves", starting with high-precision rules and progressively increasing recall. Each sieve builds on the output of its predecessor, so that it is possible to consider entity-level information. For example, in the case of {*Hillary Clinton* ←

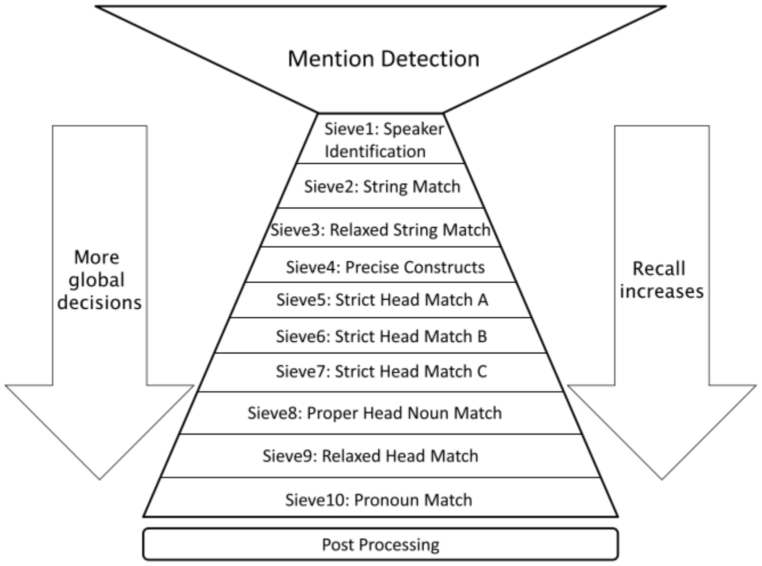

Mention Detection
Sieve1: Speaker Identification
Sieve2: String Match
Sieve3: Relaxed String Match
Sieve4: Precise Constructs
Sieve5: Strict Head Match A
Sieve6: Strict Head Match B
Sieve7: Strict Head Match C
Sieve8: Proper Head Noun Match
Sieve9: Relaxed Head Match
Sieve10: Pronoun Match
Post Processing
More global decisions
Recall increases

Figure 15.5: Architecture of Stanford's deterministic "multi-pass sieve" coreference system (Lee et al., 2013)

Clinton $\leftarrow$ *she*}, the name-matching sieve would link *Clinton* and *Hillary Clinton*, and the pronoun-matching sieve would then link *she* to the combined cluster.

The Stanford deterministic system made a strong showing at 2011 CoNLL shared task on coreference, winning nearly every track in the competition (Pradhan et al., 2011). This was particularly surprising, given the dominance of non-deterministic methods based on machine learning in virtually all other areas of natural language processing. While learning-based systems have regained the upper hand in recent years (e.g., Björkelund and Kuhn, 2014; **?**), the resurgence of deterministic, rule-based artificial intelligence in coreference resolution tells us that it may differ in important ways from other tasks, such as tagging and dependency parsing.

15.3 Entity linking and multi-document coreference resolution

[todo: later]

Exercises

1. The size of the largest entity typically grows linearly with the number of mentions in a document. Using this assumption, give an asymptotic estimate of the num-

ber of antecedent structures that are compatible with a coreference clustering in a document with M mentions.

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.

Part IV

Applications

Chapter 17

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 13. 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.

17.1 Entities

The starting point for information extraction is to identify mentions of entities in text. For example, consider the following text.

(17.1) *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**.

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**.

17.1.1 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*

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 17.1: BIO notation for named entity recognition

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 17.1. 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 17.1. 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 17.1 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).

17.1.2 Alternative modeling frameworks*

Apart from sequence labeling, there are other formulations for named entity recognition, which are arguably better customized for the task.

17.2 Relations

17.2.1 Knowledge-base population

17.2.2 Distant supervision

17.3 Events and processes

17.4 Facts, beliefs, and hypotheticals

Chapter 18

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.

Throughout the course, we've been working with the general formulation,

$$\hat{\boldsymbol{y}} = \operatorname*{argmax}_{\boldsymbol{y} \in \mathcal{Y}} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{y}). \tag{18.1}$$

Now suppose we make $\boldsymbol{x}$ a sentence in a foreign (**source**) language, and $\boldsymbol{y} \in \mathcal{Y}$ a sentence in the **target language**. We can thus view translation in the same linear structure-prediction formalism that we have used for tasks like tagging and parsing. This formalism requires two main algorithms: an **estimation** algorithm for computing the parameters $\boldsymbol{\theta}$, and a **decoding** algorithm for computing $\hat{\boldsymbol{y}}$. Machine translation poses unique challenges for both of these algorithms.

Estimation is complicated because we typically receive supervision in the form of **bitext**, or aligned sentences, e.g.,

$$\boldsymbol{x} = \textit{A Vinay le gusta las manzanas.}$$
$$\boldsymbol{y} = \textit{Vinay likes apples.}$$

A useful feature function would note the translation pairs $\langle gusta, likes \rangle$, $\langle manzanas, apples \rangle$, and even $\langle Vinay, Vinay \rangle$. But this word-to-word **alignment** is not given in the data. One solution is to treat this alignment as a **latent variable**; this is the approach taken by classical **statistical machine translation** (SMT) systems, described in § 18.1. Another solution is to model the relationship between $\boldsymbol{x}$ and $\boldsymbol{y}$ through a more complex and expressive function; this is the approach taken by **neural machine translation** (NMT) systems, described in § 18.2.

Decoding is also difficult for machine translation, because of the huge space of possible translations, $\mathcal{Y}$. We have faced large label spaces before: for example, in sequence labeling, the set of possible label sequences is exponential in the length of the input. In these cases, it was possible to search the space quickly by introducing locality assumptions: for example, that a single tag depends only on its predecessor, or that a single production depends only on its parent. In machine translation, no such locality assumptions seem to be possible: human translators reword, reorder, and rearrange words at will; they replace single words with multi-word phrases, and vice versa. This flexibility means that in even relatively simple translation models, decoding is NP-hard (Knight, 1999). Approaches for dealing with this complexity are described in § 18.3.

18.1 Statistical machine translation in the noisy channel model

There are two major criteria for a translation:

- **Adequacy**: The translation $\hat{\boldsymbol{y}}$ should adequately reflect the linguistic content of $\boldsymbol{w}$. For example, if $\boldsymbol{x}$ = *A Vinay le gusta Python*, the *gloss*[1] $\boldsymbol{y}$ = *To 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.
- **Fluency**: The translation $\hat{\boldsymbol{y}}$ should read like fluent text in the target language. By this criterion, the gloss $\boldsymbol{y}$ = *To Vinay it like Python* will score poorly, and $\boldsymbol{y}$ = *Vinay likes Python* will be preferred.

	Adequate?	Fluent?
To Vinay it like Python	yes	no
Vinay debugs memory leaks	no	yes
Vinay likes Python	yes	yes

Table 18.1: Adequacy and fluency for translations of the Spanish *A Vinay le gusta Python*

An early insight in machine translation was that the scoring function for a translation can decompose across these criteria:

$$\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{\theta}_t \cdot \boldsymbol{f}_t(\boldsymbol{x}, \boldsymbol{y}) + \boldsymbol{\theta}_\ell \cdot \boldsymbol{f}_\ell(\boldsymbol{y}) \tag{18.2}$$

[1]A "gloss" is a word-for-word translation.

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 $\boldsymbol{\theta}_\ell$ 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

$$\boldsymbol{f}_\ell(\boldsymbol{y}) = \bigcup_i \{\boldsymbol{y}_{i:(i+k)}\} \tag{18.3}$$

$$\theta_\ell(\{y_i, y_{i+1}, \ldots, y_{i+k}\}) = \log \mathrm{p}(y_{i+k} \mid y_i, y_{i+1}, \ldots, y_{i+k-1}) \tag{18.4}$$

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 many 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,

$$\boldsymbol{\theta}_\ell \cdot \boldsymbol{f}_\ell(\boldsymbol{y}) = \log \mathrm{p}(\boldsymbol{y}). \tag{18.5}$$

Now let's consider the translation component. If we can set

$$\boldsymbol{\theta}_t \cdot \boldsymbol{f}_t(\boldsymbol{y}, \boldsymbol{x}) = \log \mathrm{p}(\boldsymbol{x} \mid \boldsymbol{y}), \tag{18.6}$$

then the sum of these two scores yields,

$$\begin{aligned} \boldsymbol{\theta}_t \cdot \boldsymbol{f}_t(\boldsymbol{y}, \boldsymbol{x}) + \boldsymbol{\theta}_\ell \cdot \boldsymbol{f}_\ell(\boldsymbol{y}) &= \log \mathrm{p}(\boldsymbol{x} \mid \boldsymbol{y}) + \log \mathrm{p}(\boldsymbol{y}) && (18.7)\\ &= \log \mathrm{p}(\boldsymbol{x}, \boldsymbol{y}). && (18.8) \end{aligned}$$

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,

$$\operatorname*{argmax}_{\boldsymbol{y}} \mathrm{p}(\boldsymbol{y} \mid \boldsymbol{x}) = \operatorname*{argmax}_{\boldsymbol{y}} \frac{\mathrm{p}(\boldsymbol{y}, \boldsymbol{x})}{\mathrm{p}(\boldsymbol{x})}, \tag{18.9}$$

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 (or whatever is the target language), and is modeled as drawn from a source language model $\boldsymbol{y} \sim \mathrm{p}_\ell(\boldsymbol{y})$
- The original text was then stochastically encoded, according to the translation model, $\boldsymbol{x} \mid \boldsymbol{y} \sim \mathrm{p}_t(\boldsymbol{x} \mid \boldsymbol{y})$.
- If we can estimate the stochastic processes p_ℓ and p_t, we can reverse the cipher and obtain the original text.

18.1.1 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 can be viewed as a grand challenge for natural language technology.

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. For example, given $\boldsymbol{x} =$ *A Vinay le gusta Python* and $\boldsymbol{y} =$ *Vinay likes Python*, one possible word-to-word alignment is,

$$\mathcal{A}(\boldsymbol{x}, \boldsymbol{y}) = \{\langle Vinay, Vinay\rangle, \langle gusta, likes\rangle, \langle Python, Python\rangle, \langle A, \varnothing\rangle, \langle le, \varnothing\rangle\}. \tag{18.10}$$

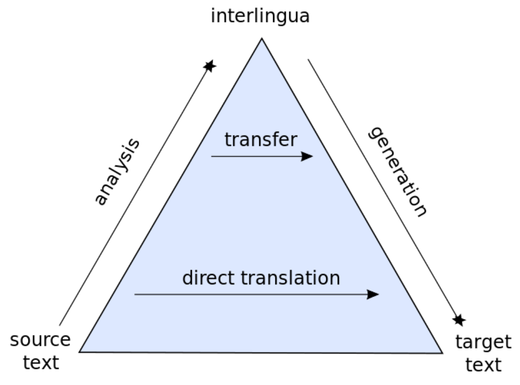


Figure 18.1: The Vauquois Pyramid ("Direct translation and transfer translation pyramid". Licensed under Creative Commons Attribution-Share Alike 3.0 via Wikimedia Commons.)

Another, less promising, possiblity is:

$$\mathcal{A}(\boldsymbol{x}, \boldsymbol{y}) = \{\langle A, Vinay\rangle, \langle Vinay, likes\rangle, \langle le, Python\rangle, \langle gusta, \varnothing\rangle, \langle Python, \varnothing\rangle\}. \tag{18.11}$$

Given the alignment, we can define the translation probability as,

$$\mathrm{p}_t(\boldsymbol{x}, \mathcal{A} \mid \boldsymbol{y}) = \prod_i \mathrm{p}(x_i, a_i \mid y_{a_i}) \tag{18.12}$$

$$= \prod_i \mathrm{p}_a(a_i \mid i, N_x, N_y) \times \mathrm{p}_{x|y}(x_i \mid y_{a_i}). \tag{18.13}$$

This probability model makes some assumptions that we now state explicitly:

- 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 $\mathrm{p}(\boldsymbol{x} \mid \boldsymbol{y}, \mathcal{A}) = \prod_i \mathrm{p}_{x|y}(x_i \mid y_{a_i})$, which means that each word in $\boldsymbol{x}$ depends only on its aligned word in $\boldsymbol{y}$. This means that 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 relaxed 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:

$$\mathrm{p}_a(a_i \mid i, N_x, N_y) = \frac{1}{N_y} \tag{18.14}$$

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

$$\hat{\boldsymbol{y}} = \operatorname*{argmax}_{\boldsymbol{y}} \mathrm{p}(\boldsymbol{x}, \boldsymbol{y}) \tag{18.15}$$

$$= \operatorname*{argmax}_{\boldsymbol{y}} \sum_{\mathcal{A}} \mathrm{p}(\boldsymbol{x}, \boldsymbol{y}, \mathcal{A}) \tag{18.16}$$

$$= \operatorname*{argmax}_{\boldsymbol{y}} \mathrm{p}_{\ell}(\boldsymbol{y}) \sum_{\mathcal{A}} \mathrm{p}_t(\boldsymbol{x}, \mathcal{A} \mid \boldsymbol{y}) \tag{18.17}$$

Conversely, if we had an accurate translation model, we could estimate beliefs about each alignment decision,

$$q_i(a_i \mid \boldsymbol{x}, \boldsymbol{y}) \propto \mathrm{p}_a(a_i \mid i, N_x, N_y) \times \mathrm{p}_{x|y}(\boldsymbol{x}_i \mid \boldsymbol{y}_{a_i}), \tag{18.18}$$

where $q_i(a_i \mid \boldsymbol{x}, \boldsymbol{y})$ is the "belief" about the alignment for word x_i.

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,

$$q_i(a_i) \propto \mathrm{p}_a(a_i \mid i, N_x, N_y)\mathrm{p}_{x|y}(\boldsymbol{x}_i \mid \boldsymbol{y}_{a_i}) \tag{18.19}$$

M-step Update the translation model,

$$\theta_{u \to v} = \log \frac{\sum_i \sum_j q_i(a_i = j)\delta(y_j = u \wedge x_i = v)}{\sum_i \sum_j q_i(a_i = j)\delta(y_j = u)} \tag{18.20}$$

Example for IBM Model 1

Suppose we have an English/French bilingual text (**bitext**) with two sentence pairs:

(18.1) *The coffee*
Le cafe

(18.2) *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	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$

Now suppose we want to translate from $\boldsymbol{x}$ = French to $\boldsymbol{y}$ = English. In the E-step, we compute alignment probabilities for each sentence. We start with the vector of alignment probabilities for the first word in the first example, $x_0 = le$.

$$q_0(0) \propto \mathrm{p}_a(0) \times \mathrm{p}(le \mid the) = \frac{1}{2} \times \frac{1}{3} \tag{18.21}$$

$$q_0(1) \propto \mathrm{p}_a(1) \times \mathrm{p}(le \mid coffee) = \frac{1}{2} \times \frac{1}{3} \tag{18.22}$$

$$q_0(\cdot) = \left[\frac{1}{2}, \frac{1}{2}\right] \tag{18.23}$$

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:

$$\mathrm{p}_{x|y}(le \mid the) = \frac{\sum_{i,j} q_i(j)\delta(x_i = le \wedge y_j = the)}{\sum_{i,j} q_i(j)\delta(y_j = the)} = \frac{\frac{1}{2}}{\frac{1}{2}+\frac{1}{2}} = \frac{1}{2} \tag{18.24}$$

$$\mathrm{p}_{x|y}(cafe \mid the) = \frac{\sum_{i,j} q_i(j)\delta(x_i = le \wedge y_j = the)}{\sum_{i,j} q_i(j)\delta(y_j = the)} = \frac{\frac{1}{2}}{\frac{1}{2}+\frac{1}{2}} = \frac{1}{2} \tag{18.25}$$

$$\mathrm{p}_{x|y}(mon \mid the) = \frac{\sum_{i,j} q_i(j)\delta(x_i = le \wedge y_j = the)}{\sum_{i,j} q_i(j)\delta(y_j = the)} = \frac{0}{\frac{1}{2}+\frac{1}{2}} = 0 \tag{18.26}$$

The math works out similarly for $\mathrm{p}(\cdot \mid my)$. But the English word *coffee* appears in both sentence pairs, so:

$$\mathrm{p}_{x|y}(le \mid coffee) = \frac{\frac{1}{2}}{4 \times \frac{1}{2}} = \frac{1}{4} \tag{18.27}$$

$$\mathrm{p}_{x|y}(cafe \mid coffee) = \frac{2 \times \frac{1}{2}}{4 \times \frac{1}{2}} = \frac{1}{2} \tag{18.28}$$

$$\mathrm{p}_{x|y}(mon \mid coffee) = \frac{\frac{1}{2}}{4 \times \frac{1}{2}} = \frac{1}{4} \tag{18.29}$$

$$\tag{18.30}$$

To summarize the new translation probabilities:

	le	*mon*	*cafe*
the	$\frac{1}{2}$	0	$\frac{1}{2}$
my	0	$\frac{1}{2}$	$\frac{1}{2}$
coffee	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{2}$

We now go back to the E-step and compute the alignments again.

$$q_0(0) \propto \mathrm{p}_a(0) \times \mathrm{p}(\textit{le} \mid \textit{the}) = \frac{1}{2} \times \frac{1}{2} \tag{18.31}$$

$$q_0(1) \propto \mathrm{p}_a(1) \times \mathrm{p}(\textit{le} \mid \textit{coffee}) = \frac{1}{2} \times \frac{1}{4} \tag{18.32}$$

$$q_0(\cdot) = \left[\frac{2}{3}, \frac{1}{3}\right] \tag{18.33}$$

$$q_1(0) \propto \mathrm{p}_a(0) \times \mathrm{p}(\textit{le} \mid \textit{coffee}) = \frac{1}{2} \times \frac{1}{4} \tag{18.34}$$

$$q_1(1) \propto \mathrm{p}_a(1) \times \mathrm{p}(\textit{cafe} \mid \textit{coffee}) = \frac{1}{2} \times \frac{1}{2} \tag{18.35}$$

$$q_1(\cdot) = \left[\frac{1}{3}, \frac{2}{3}\right] \tag{18.36}$$

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:

$$\mathrm{p}_{x|y}(\textit{le} \mid \textit{the}) = \frac{\sum_{i,j} q_i(j)\delta(x_i = \textit{le} \wedge y_j = \textit{the})}{\sum_{i,j} q_i(j)\delta(y_j = \textit{the})} = \frac{\frac{2}{3}}{\frac{2}{3} + \frac{1}{3}} = \frac{2}{3} \tag{18.37}$$

$$\mathrm{p}_{x|y}(\textit{cafe} \mid \textit{the}) = \frac{\sum_{i,j} q_i(j)\delta(x_i = \textit{le} \wedge y_j = \textit{the})}{\sum_{i,j} q_i(j)\delta(y_j = \textit{the})} = \frac{\frac{1}{3}}{\frac{1}{3} + \frac{2}{3}} = \frac{1}{3} \tag{18.38}$$

$$\mathrm{p}_{x|y}(\textit{mon} \mid \textit{the}) = 0 \tag{18.39}$$

$$\mathrm{p}_{x|y}(\textit{le} \mid \textit{coffee}) = \frac{\frac{1}{3}}{\frac{1}{3} + \frac{2}{3} + \frac{1}{3} + \frac{2}{3}} = \frac{1}{6} \tag{18.40}$$

$$\mathrm{p}_{x|y}(\textit{cafe} \mid \textit{coffee}) = \frac{2 \times \frac{2}{3}}{2} = \frac{2}{3} \tag{18.41}$$

$$\mathrm{p}_{x|y}(\textit{mon} \mid \textit{coffee}) = \frac{\frac{1}{3}}{2} = \frac{1}{6} \tag{18.42}$$

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

	le	*mon*	*cafe*
the	$\frac{2}{3}$	0	$\frac{1}{3}$
my	0	$\frac{2}{3}$	$\frac{1}{3}$
coffee	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{2}{3}$

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,

$$\mathrm{p}_a(a_i; i, N_x, N_y) = \phi_{a_i,i,N_x,N_y} \tag{18.43}$$

$$s.t. \forall i, N_x, N_y, \sum_a \phi_{a,i,N_x,N_y} = 1. \tag{18.44}$$

The variables a_i and i are integer indices, so ϕ_{a,i,N_x,N_y} represents the probability that token i is aligned to token a_i in sentence pairs with length N_x and N_y. We compute this probability by the relative frequency estimate,

$$\phi_{a,i,N_x,N_y} = \frac{\sum_{\boldsymbol{y},\boldsymbol{x}:\#|\boldsymbol{y}|=N_y,\#|\boldsymbol{x}|=N_x} q_i(a)}{\sum_{\boldsymbol{y},\boldsymbol{x}:\#|\boldsymbol{y}|=N_y,\#|\boldsymbol{x}|=N_x} \delta(\#|\boldsymbol{y}| < i)}, \tag{18.45}$$

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 is 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 in Spanish:

(18.3) *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,

$$\mathrm{p}_{x|y,\mathcal{A}}(\textit{daba una bofetada} \mid \textit{slap}) \tag{18.46}$$

$$= \mathrm{p}_{x|y}(\textit{daba} \mid \textit{slap}) \times \mathrm{p}_{x|y}(\textit{una} \mid \textit{slap}) \times \mathrm{p}_{x|y}(\textit{bofetada} \mid \textit{slap}). \tag{18.47}$$

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.

18.1.2 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:

(18.4) *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:

(18.5) *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 multi-word units; alignments are mappings from spans to spans, $\langle (i,j),(k,\ell)\rangle$, so that

$$\mathrm{p}(\boldsymbol{x} \mid \boldsymbol{y}, \mathcal{A}) = \prod_{\langle (i,j),(k,\ell)\rangle \in \mathcal{A}} \mathrm{p}_{x|y}(\{x_i, x_{i+1}, \ldots, x_j\} \mid \{y_k, y_{k+1}, \ldots, y_\ell\}), \tag{18.48}$$

where we require that the alignment set $\mathcal{A}$ cover both sentences with non-overlapping spans, as shown in **??**. [todo: add figure]

18.2 Neural machine translation

The statistical paradigm for machine translation relies on decoupling the translation problem into modular components: language modeling for target language fluency, an alignment model to link words or phrases across the source and target, and then a translation table to compute translation probabilities under a given alignment. The advantage of this approach is that each component can be relatively simple, and can reuse techniques from other areas of NLP: for example, we can use the same basic language modeling technology in translation as in speech recognition (chapter 5); the expectation-maximization algorithm for alignment can be reused from semi-supervised learning (chapter 4). However,

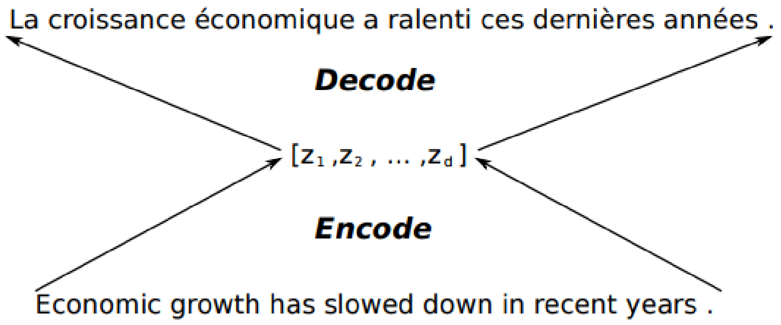

La croissance économique a ralenti ces dernières années .
Decode
[z₁ ,z₂ , ... ,z_d]
Encode
Economic growth has slowed down in recent years .

Figure 18.2: Schematic of the encoder-decoder architecture (Cho et al., 2014a)

this modularity also has downsides. First, a combination of simple modules may simply not be expressive enough for the translation task. For example, the assumption that each phrase-to-phrase translation decision is independent will fail to capture thematic, stylistic, and topical dependencies across the translation. Second, it is difficult to train each of the translation modules appropriately.[todo: say more about this]

Neural machine translation is a relatively recent innovation, which replaces this modular design with an integrated, end-to-end architecture. The inner workings of this architecture are not alignments between words or phrases, but rather, vector encodings of the source-language text and the ongoing translation into the target language. Because each component is differentiable, the entire architecture can be trained by backpropagation from a translation error signal.

While neural machine translation is a rapidly evolving area of research, there are some basic principles that unite many of the contemporary approaches. The first is to model the generation of target-language text as a **conditional recurrent neural network**. This means that there is a recurrent update to a hidden state vector, which in turn is used to generate the next token in the output sequence. Both the generation and the recurrent update may be conditioned on an additional vector of information, which encodes the meaning of the source sentence. The combination of encoding and decoding into a single model is called an **encoder-decoder architecture** (Cho et al., 2014b), shown in Figure 18.2.

18.2.1 Sequence-to-sequence translation

A relatively simple example of an encoder-decoder architecture is the sequence-to-sequence model of Sutskever et al. (2014). In this model, the encoder is a **long short-term memory** (LSTM chapter 5), which computes a vector $\boldsymbol{z}$ from the final state of the source language input $\boldsymbol{x}$. This vector is then used to help generate the target language output,

$$\boldsymbol{h}_m = \text{LSTM-UPDATE}(E^{(x)}_{x_m}, \boldsymbol{h}_{m-1}) \tag{18.49}$$

$$\boldsymbol{z}_n = \text{LSTM-UPDATE}([E^{(y)}_{y_n}; \boldsymbol{h}_M], \boldsymbol{z}_{n-1}) \tag{18.50}$$

$$y_{n+1} \sim \text{SOFTMAX}(\mathbf{U}\boldsymbol{z}_n). \tag{18.51}$$

In these equations $E^{(x)}$ is a matrix of embeddings for the source language words, $E^{(y)}$ is a matrix of embeddings for the target language models, and $[E^{(y)}_{y_n}; \boldsymbol{h}_M]$ is a vertical concatenation of the embedding for target language word y_n and the encoding of the source-language input $\boldsymbol{h}_M$, which is simply the final hidden state in the encoder LSTM. The target-language hidden state is then left-multiplied by a matrix of output embeddings, $\mathbf{U}$, and the product is passed through a softmax transformation, giving a distribution over target language tokens.

A key point about this model is that the source text is encoded into a single, fixed-length vector, which is equal to the final state in the encoder LSTM.[2] This means that information from the earlier part of the source sentence may be attenuated by repeated applications of the LSTM update.[3] Sutskever *et al.* address this issue by reversing the source text, so that it is read from the end to the beginning. Nonetheless, the model is surprisingly effective, competing with some of the English-to-French translation systems that were available at the time.

While the LSTM output model is simple, decoding still requires approximate search. Each output token y_n affects the recurrent state $\boldsymbol{z}_n$, so that the optimal local choice at position n could have negative consequences later in the translation. These issues are discussed in § 18.3.

18.2.2 Neural attention for machine translation

A surprising aspect of the sequence-to-sequence model is that it makes no attempt to link words or phrases across the source and target texts. This would seem to pose problems in translating long sentences, where many words are crammed into a fixed encoding of the source text.

Neural attention is a solution, which integrates aspects of the "alignment" concept from statistical machine translation into a neural translation architecture. The key idea is to compute a variable-length encoding of the source text, such as the sequence of hidden states across the encoding process, $[\boldsymbol{h}_1, \boldsymbol{h}_2, \ldots, \boldsymbol{h}_M]$. At each token n in the target language output, we compute an **attention vector** over this variable-length source language encoding, $\boldsymbol{\alpha}_n$. Examples of the attention vectors across sentence pairs are shown in Figure 18.3.

[2]A related approach is to use a **convolutional neural network** to encode the source text into a fixed representation, and then use a sequence model such as the LSTM to decode (Kalchbrenner and Blunsom,

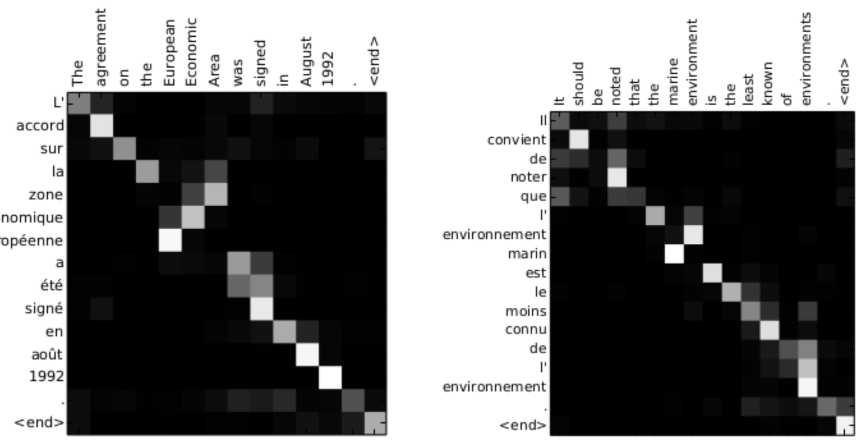


Figure 18.3: Neural attention between the source and target texts, from Bahdanau et al. (2014). Each French word, shown on the rows, has an attention vector over the words in the English-language source.

We can then compute the context vector as the weighted sum,

$$\boldsymbol{c}_n = \sum_{m=1}^{M} \alpha_{m\to n} \boldsymbol{h}_m. \tag{18.52}$$

This context vector can then be incorporated into the output LSTM, similar to how $\boldsymbol{h}_M$ was used in the sequence-to-sequence model (Equation 18.51).

The attention vectors are themselves computed as a function of the source encodings $\{\boldsymbol{h}_m\}$ and the current hidden state of the decoding model, $\boldsymbol{z}_n$. Bahdanau et al. (2014) propose the following formulation:

$$a_{m\to n} = \boldsymbol{v}^\top \tanh\left(\mathbf{W}_a z_n + \mathbf{U}_a \boldsymbol{h}_m\right) \tag{18.53}$$

$$\boldsymbol{\alpha}_{\to n} = \text{SoftMax}(\boldsymbol{a}_{\to n}) \tag{18.54}$$

where $\boldsymbol{v} \in \mathbb{R}^K$ is a parameter vector, and $\mathbf{W}_a$ and $\mathbf{U}_a$ are parameter matrices. The vector $\tanh\left(\mathbf{W}_a z_n + \mathbf{U}_a \boldsymbol{h}_m\right)$ can be viewed as the hidden layer of a feedforward neural network, which combines aspects of the source and target text. The vector $\boldsymbol{v}$ then projects this hidden layer to a single score for each pair (m, n). These scores are then passed through a SoftMax activation to create a probability vector over indices m in the source.

Bahdanau et al. (2014) use the context vector $\boldsymbol{c}_n = \sum_{m=1}^{M} \alpha_{m\to n} \boldsymbol{h}_m$ in two ways. First, they use it within the recurrent update of the decoder hidden state $\boldsymbol{z}_n$, which depends

2013).

[3]The effect of this attenuation is much milder for the LSTM than for traditional recurrent neural networks (RNNs), due to the use of the memory cell, which can "remember" information over several steps.

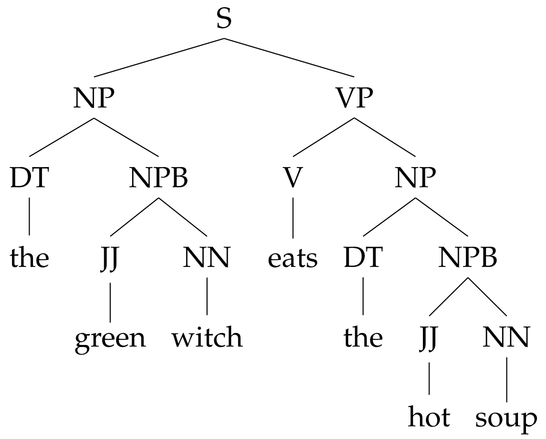
on $\boldsymbol{c}_n$ as well as on the previous state $\boldsymbol{z}_{n-1}$ and the emission y_n. Second, they use it in computing the probability over the output y_{n+1}, which is conditioned on the hidden state $\boldsymbol{z}_n$, the context $\boldsymbol{c}_n$, and also the previous output y_n. The computation graph is shown in **??**.

18.3 Decoding

In general, decoding works by incremental search on multiple beams. Each beam represents a potential translation path.

18.4 *Syntactic MT

Consider the English sentence, *The green witch eats the hot soup.*



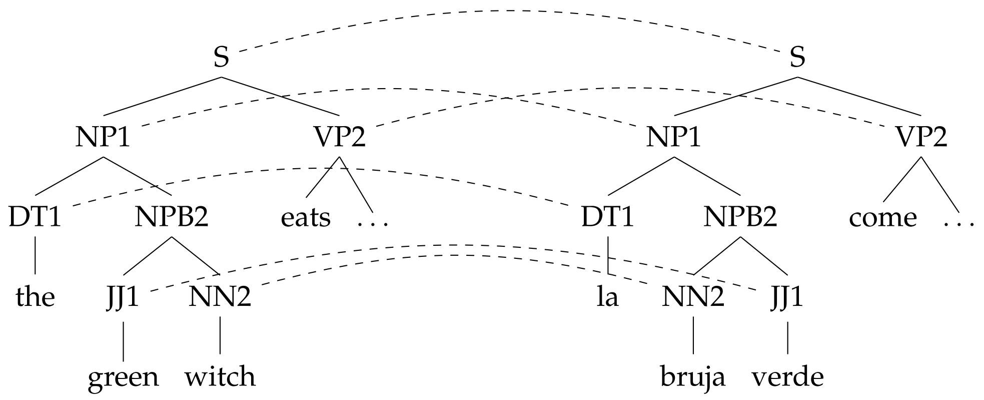
Where NPB is a "bare NP," without the determiner. We might get this non-terminal from binarizing a CFG.

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.

18.4.1 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:

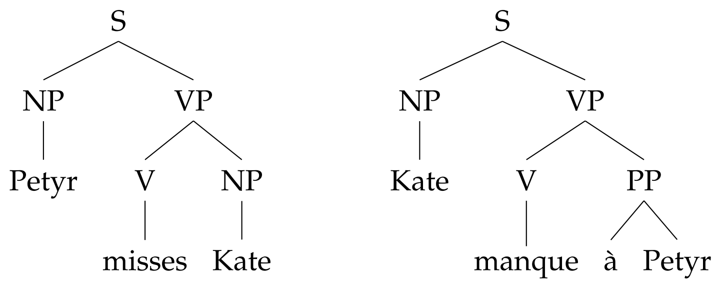
$$\begin{aligned} A &\to B\ C\ D\ E, & &C\ E\ B\ D \\ A &\to [B\ C]\ D\ E, & &[C\ E\ B]\ D \\ A &\to B\ [C\ D]\ E, & &[C\ E\ B\ D] \\ A &\to B\ C\ [D\ E], & &C\ [E\ B\ D] \end{aligned}$$

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.

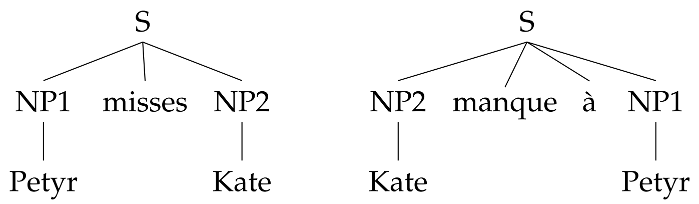
18.4.2 No raising or lowering

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.

18.4.3 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

$NP \rightarrow D\ NPB,$	$D\ NPB$	1.0
$NPB \rightarrow N_1\ J_2,$	$J_2\ N_1$	0.8
$NPB \rightarrow N_1\ N_2,$	$N_1\ N_2$	0.2
$D \rightarrow$ *la*,	*the*	0.5
$N \rightarrow$ *razon*,	*reason*	0.5
$N \rightarrow$ *principal*,	*principal*	0.5
$J \rightarrow$ *principal*,	*main*	1.0

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

$$\psi(X, i, j, i', j') = \max_{k,k',A,B} P(S \rightarrow A\ B, A\ B) \otimes \psi(A, i, k, i', k') \otimes \psi(B, k, j, k', j')$$
$$\oplus P(S \rightarrow A\ B, B\ A) \otimes \psi(A, i, k, k', j') \otimes \psi(B, k, j, i', k')$$

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.

Bibliography

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I. J., Harp, A., Irving, G., Isard, M., Jia, Y., Józefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D. G., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P. A., Vanhoucke, V., Vasudevan, V., Viégas, F. B., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2016). Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467.

Abend, O. and Rappoport, A. (2017). The state of the art in semantic representation. In *Proceedings of the Association for Computational Linguistics (ACL)*, Vancouver.

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.

Alm, C. O., Roth, D., and Sproat, R. (2005). Emotions from text: machine learning for text-based emotion prediction. In *Proceedings of Empirical Methods for Natural Language Processing (EMNLP)*, pages 579–586.

Aluísio, S., Pelizzoni, J., Marchi, A., de Oliveira, L., Manenti, R., and Marquiafável, V. (2003). An account of the challenge of tagging a reference corpus for brazilian portuguese. *Computational Processing of the Portuguese Language*, pages 194–194.

Anand, P., Walker, M., Abbott, R., Fox Tree, J. E., Bowmani, R., and Minor, M. (2011). Cats rule and dogs drool!: Classifying stance in online debate. In *Proceedings of the 2nd Workshop on Computational Approaches to Subjectivity and Sentiment Analysis (WASSA 2.011)*, pages 1–9, Portland, Oregon. Association for Computational Linguistics.

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.

Andor, D., Alberti, C., Weiss, D., Severyn, A., Presta, A., Ganchev, K., Petrov, S., and Collins, M. (2016). Globally normalized transition-based neural networks. In *Proceedings of the Association for Computational Linguistics (ACL)*, pages 2442–2452, Berlin.

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.

Artstein, R. and Poesio, M. (2008). Inter-coder agreement for computational linguistics. *Computational Linguistics*, 34(4):555–596.

Artzi, Y. and Zettlemoyer, L. (2013). Weakly supervised learning of semantic parsers for mapping instructions to actions. *Transactions of the Association for Computational Linguistics*, 1:49–62.

Aw, A., Zhang, M., Xiao, J., and Su, J. (2006). A phrase-based statistical model for SMS text normalization. In *Proceedings of the Association for Computational Linguistics (ACL)*, pages 33–40.

Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. In *Neural Information Processing Systems (NIPS)*, Montréal.

Banarescu, L., Bonial, C., Cai, S., Georgescu, M., Griffitt, K., Hermjakob, U., Knight, K., Koehn, P., Palmer, M., and Schneider, N. (2013). Abstract meaning representation for sembanking. In *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse*, pages 178–186, Sofia, Bulgaria. Association for Computational Linguistics.

Banko, M., Cafarella, M. J., Soderland, S., Broadhead, M., and Etzioni, O. (2007). Open information extraction from the web. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, volume 7, pages 2670–2676.

Barnickel, T., Weston, J., Collobert, R., Mewes, H.-W., and Stümpflen, V. (2009). Large scale application of neural network based semantic role labeling for automated relation extraction from biomedical texts. *PLoS One*, 4(7):e6393.

Baron, A. and Rayson, P. (2008). Vard2: A tool for dealing with spelling variation in historical corpora. In *Postgraduate conference in corpus linguistics*.

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.

Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *Neural Networks, IEEE Transactions on*, 5(2):157–166.

Bengtson, E. and Roth, D. (2008). Understanding the value of features for coreference resolution. In *Proceedings of Empirical Methods for Natural Language Processing (EMNLP)*, pages 294–303, Honolulu, HI.

Benjamini, Y. and Hochberg, Y. (1995). Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 289–300.

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.

Berg-Kirkpatrick, T., Burkett, D., and Klein, D. (2012). An empirical investigation of statistical significance in nlp. In *Proceedings of Empirical Methods for Natural Language Processing (EMNLP)*, pages 995–1005.

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.

Bhatia, P., Guthrie, R., and Eisenstein, J. (2016). Morphological priors for probabilistic neural word embeddings. In *Proceedings of Empirical Methods for Natural Language Processing (EMNLP)*.

Biber, D. (1991). *Variation across speech and writing*. Cambridge University Press.

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.

Björkelund, A. and Kuhn, J. (2014). Learning structured perceptrons for coreference resolution with latent antecedents and non-local features. In *Proceedings of the Association for Computational Linguistics (ACL)*, pages 47–57, Baltimore, MD.

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.

Blei, D. M., Ng, A. Y., and Jordan, M. I. (2003). Latent dirichlet allocation. *the Journal of machine Learning research*, 3:993–1022.

Böhmová, A., Hajič, J., Hajičová, E., and Hladká, B. (2003). The prague dependency treebank. In *Treebanks*, pages 103–127. Springer.

Bollacker, K., Evans, C., Paritosh, P., Sturge, T., and Taylor, J. (2008). Freebase: a collaboratively created graph database for structuring human knowledge. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1247–1250. AcM.

Bontcheva, K., Dimitrov, M., Maynard, D., Tablan, V., and Cunningham, H. (2002). Shallow methods for named entity coreference resolution. In *Chaınes de références et résolveurs danaphores, workshop TALN*.

Botha, J. A. and Blunsom, P. (2014). Compositional morphology for word representations and language modelling. In *Proceedings of the International Conference on Machine Learning (ICML)*.

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.

Branavan, S., Chen, H., Eisenstein, J., and Barzilay, R. (2009). Learning document-level semantic properties from free-text annotations. *Journal of Artificial Intelligence Research*, 34(2):569.

Brants, T. (2000). Tnt: a statistical part-of-speech tagger. In *Proceedings of the sixth conference on Applied natural language processing*, pages 224–231. Association for Computational Linguistics.

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.

Brun, C. and Roux, C. (2014). Décomposition des "hash tags" pour l'amélioration de la classification en polarité des "tweets". *Proceedings of Traitement Automatique des Langues Naturelles*, pages 473–478.

Cai, Q. and Yates, A. (2013). Large-scale semantic parsing via schema matching and lexicon extension. In *Proceedings of the Association for Computational Linguistics (ACL)*, Sophia, Bulgaria.

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.

Cardie, C. and Wagstaff, K. (1999). Noun phrase coreference as clustering. In *EMNLP*, pages 82–89.

Carletta, J. (1996). Assessing agreement on classification tasks: the kappa statistic. *Computational linguistics*, 22(2):249–254.

Carpenter, B. (1997). *Type-logical semantics*. MIT press.

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.

Carreras, X. and Màrquez, L. (2005). Introduction to the conll-2005 shared task: Semantic role labeling. In *Proceedings of the Ninth Conference on Computational Natural Language Learning*, pages 152–164. Association for Computational Linguistics.

Carroll, L. (1917). *Through the looking glass: And what Alice found there*. Rand, McNally.

Chang, M.-W., Ratinov, L.-A., Rizzolo, N., and Roth, D. (2008). Learning and inference with constraints. In *AAAI*, pages 1513–1518.

Charniak, E. (1997). Statistical techniques for natural language parsing. *AI magazine*, 18(4):33.

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.

Chen, T. and Guestrin, C. (2016). Xgboost: A scalable tree boosting system. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 785–794. ACM.

Cho, K. (2015). Natural language understanding with distributed representation. *CoRR*, abs/1511.07916.

Cho, K., Van Merriënboer, B., Bahdanau, D., and Bengio, Y. (2014a). On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014b). Learning phrase representations using rnn encoder-decoder for statistical machine translation. In *Proceedings of Empirical Methods for Natural Language Processing (EMNLP)*.

Christensen, J., Soderland, S., Etzioni, O., et al. (2010). Semantic role labeling for open information extraction. In *Proceedings of the NAACL HLT 2010 First International Workshop on Formalisms and Methodology for Learning by Reading*, pages 52–60. Association for Computational Linguistics.

Chu, Y.-J. and Liu, T.-H. (1965). On shortest arborescence of a directed graph. *Scientia Sinica*, 14(10):1396.

Chung, C. and Pennebaker, J. W. (2007). The psychological functions of function words. *Social communication*, pages 343–359.

Church, K. W. (2000). Empirical estimates of adaptation: the chance of two Noriegas is closer to $p/2$ than p^2. In *Proceedings of the 18th conference on Computational linguistics-Volume 1*, pages 180–186.

Clark, K. and Manning, C. D. (2015). Entity-centric coreference resolution with model stacking. In *Proceedings of the Association for Computational Linguistics (ACL)*, pages 1405–1415, Beijing.

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.

Cohen, J. (1960). A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46.

Cohen, S. B., Gómez-Rodríguez, C., and Satta, G. (2012). Elimination of spurious ambiguity in transition-based dependency parsing. *CoRR*, abs/1206.6735.

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.

Collobert, R., Kavukcuoglu, K., and Farabet, C. (2011a). Torch7: A matlab-like environment for machine learning. Technical Report EPFL-CONF-192376, EPFL.

Collobert, R. and Weston, J. (2008). A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 160–167.

Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., and Kuksa, P. (2011b). Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12:2493–2537.

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)*.

Danescu-Niculescu-Mizil, C., Sudhof, M., Jurafsky, D., Leskovec, J., and Potts, C. (2013). A computational approach to politeness with application to social factors. In *Proceedings of the Association for Computational Linguistics (ACL)*, pages 250–259, Sophia, Bulgaria.

Das, D., Chen, D., Martins, A. F., Schneider, N., and Smith, N. A. (2014). Frame-semantic parsing. *Computational Linguistics*, 40(1):9–56.

Davidson, D. (1967). The logical form of action sentences. In Rescher, N., editor, *The Logic of Decision and Action*. University of Pittsburgh Press, Pittsburgh.

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.

Dehdari, J. (2014). *A Neurophysiologically-Inspired Statistical Language Model*. PhD thesis, The Ohio State University.

Denis, P. and Baldridge, J. (2007). A ranking approach to pronoun resolution. In *IJCAI*.

Denis, P. and Baldridge, J. (2008). Specialized models and ranking for coreference resolution. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP '08, pages 660–669, Stroudsburg, PA, USA. Association for Computational Linguistics.

Derrida, J. (1985). Des tours de babel. In Graham, J., editor, *Difference in translation*. Cornell University Press, Ithaca, NY.

Dietterich, T. G. (1998). Approximate statistical tests for comparing supervised classification learning algorithms. *Neural computation*, 10(7):1895–1923.

Dowty, D. (1991). Thematic proto-roles and argument selection. *Language*, pages 547–619.

Dredze, M., Paul, M. J., Bergsma, S., and Tran, H. (2013). Carmen: A Twitter geolocation system with applications to public health. In *AAAI workshop on expanding the boundaries of health informatics using AI (HIAI)*, pages 20–24.

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.

Efron, B. and Tibshirani, R. J. (1993). An introduction to the bootstrap: Monographs on statistics and applied probability, vol. 57. *New York and London: Chapman and Hall/CRC*.

Eisner, J. M. (1996). Three new probabilistic models for dependency parsing: An exploration. In *COLING*, pages 340–345.

Ekman, P. (1992). Are there basic emotions? *Psychological Review*, 99(3):550–553.

Esuli, A. and Sebastiani, F. (2006). Sentiwordnet: A publicly available lexical resource for opinion mining. In *LREC*, volume 6, pages 417–422. Citeseer.

Fellbaum, C. (2010). *WordNet*. Springer.

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.

Fillmore, C. J. (1976). Frame semantics and the nature of language. *Annals of the New York Academy of Sciences*, 280(1):20–32.

Fillmore, C. J. and Baker, C. (2009). A frames approach to semantic analysis. In *The Oxford Handbook of Linguistic Analysis*. Oxford University Press.

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.

Finkel, J. R. and Manning, C. D. (2008). Enforcing transitivity in coreference resolution. In *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics on Human Language Technologies: Short Papers*, pages 45–48. Association for Computational Linguistics.

Firth, J. R. (1957). *Papers in Linguistics 1934-1951*. Oxford University Press.

Flanigan, J., Thomson, S., Carbonell, J., Dyer, C., and Smith, N. A. (2014). A discriminative graph-based parser for the abstract meaning representation. In *Proceedings of the Association for Computational Linguistics (ACL)*, pages 1426–1436, Baltimore, MD.

Francis, W. N. (1964). A standard sample of present-day English for use with digial computers. Report to the U.S Office of Education on Cooperative Research Project No. E-007.

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). Relex – relation extraction using dependency parse trees. *Bioinformatics*, 23(3):365–371.

Gabrilovich, E. and Markovitch, S. (2007). Computing semantic relatedness using wikipedia-based explicit semantic analysis. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, volume 7, pages 1606–1611.

Gale, W. A., Church, K. W., and Yarowsky, D. (1992). One sense per discourse. In *Proceedings of the workshop on Speech and Natural Language*, pages 233–237. Association for Computational Linguistics.

Galley, M. (2006). A skip-chain conditional random field for ranking meeting utterances by importance. In *Proceedings of Empirical Methods for Natural Language Processing (EMNLP)*, pages 364–372.

Ganchev, K. and Dredze, M. (2008). Small statistical models by random feature mixing. In *Proceedings of the ACL08 HLT Workshop on Mobile Language Processing*, pages 19–20.

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.

Ge, R. and Mooney, R. J. (2005). A statistical semantic parser that integrates syntax and semantics. In *Proceedings of the Conference on Natural Language Learning (CoNLL)*, pages 9–16.

Geach, P. T. (1962). *Reference and generality: An examination of some medieval and modern theories*. Cornell University Press.

Gildea, D. and Jurafsky, D. (2002). Automatic labeling of semantic roles. *Computational linguistics*, 28(3):245–288.

Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep sparse rectifier networks. In *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics. JMLR W&CP Volume*, volume 15, pages 315–323.

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.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep learning*. MIT Press.

Gouws, S., Metzler, D., Cai, C., and Hovy, E. (2011). Contextual bearing on linguistic variation in social media. In *LASM*.

Graves, A., Mohamed, A.-r., and Hinton, G. (2013). Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. IEEE.

Graves, A. and Schmidhuber, J. (2005). Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural Networks*, 18(5):602–610.

Grishman, R., Macleod, C., and Sterling, J. (1992). Evaluating parsing strategies using standardized parse files. In *Proceedings of the third conference on Applied natural language processing*, pages 156–161. Association for Computational Linguistics.

Grishman, R. and Sundheim, B. (1996). Message understanding conference-6: A brief history. In *Proceedings of the International Conference on Computational Linguistics (COLING)*, volume 96, pages 466–471.

Groenendijk, J. and Stokhof, M. (1991). Dynamic predicate logic. *Linguistics and philosophy*, 14(1):39–100.

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.

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. (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.

Hayes, A. F. and Krippendorff, K. (2007). Answering the call for a standard reliability measure for coding data. *Communication methods and measures*, 1(1):77–89.

He, L., Lee, K., Lewis, M., and Zettlemoyer, L. (2017). Deep semantic role labeling: What works and what's next. In *Proceedings of the Association for Computational Linguistics (ACL)*, Vancouver.

Hearst, M. A. (1992). Automatic acquisition of hyponyms from large text corpora. In *Proceedings of the International Conference on Computational Linguistics (COLING)*, pages 539–545. Association for Computational Linguistics.

Hindle, D. and Rooth, M. (1990). Structural ambiguity and lexical relations. In *Proceedings of the Workshop on Speech and Natural Language*.

Hobbs, J. R. (1978). Resolving pronoun references. *Lingua*, 44(4):311–338.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

Hockenmaier, J. and Steedman, M. (2007). Ccgbank: a corpus of ccg derivations and dependency structures extracted from the penn treebank. *Computational Linguistics*, 33(3):355–396.

Hovy, E. and Lavid, J. (2010). Towards a 'science' of corpus annotation: a new methodological challenge for corpus linguistics. *International journal of translation*, 22(1):13–36.

Hovy, E., Marcus, M., Palmer, M., Ramshaw, L., and Weischedel, R. (2006). Ontonotes: the 90% solution. In *Proceedings of the North American Chapter of the Association for Computational Linguistics (NAACL)*, pages 57–60, New York, NY.

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, E. H., Socher, R., Manning, C. D., and Ng, A. Y. (2012a). Improving word representations via global context and multiple word prototypes. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics: Long Papers-Volume 1*, pages 873–882. Association for Computational Linguistics.

Huang, L., Fayong, S., and Guo, Y. (2012b). 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.

Huang, Z., Xu, W., and Yu, K. (2015). Bidirectional lstm-crf models for sequence tagging. *arXiv preprint arXiv:1508.01991*.

Ide, N. and Wilks, Y. (2006). Making sense about sense. In *Word sense disambiguation*, pages 47–73. Springer.

Jain, A. K. (2010). Data clustering: 50 years beyond k-means. *Pattern recognition letters*, 31(8):651–666.

Jiang, L., Yu, M., Zhou, M., Liu, X., and Zhao, T. (2011). Target-dependent twitter sentiment classification. In *Proceedings of the Association for Computational Linguistics (ACL)*, pages 151–160, Portland, OR.

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.

Jozefowicz, R., Zaremba, W., and Sutskever, I. (2015). An empirical exploration of recurrent network architectures. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 2342–2350.

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.

Kalchbrenner, N. and Blunsom, P. (2013). Recurrent continuous translation models. In *Proceedings of Empirical Methods for Natural Language Processing (EMNLP)*, pages 1700–1709.

Karlsson, F. (2007). Constraints on multiple center-embedding of clauses. *Journal of Linguistics*, 43(02):365–392.

Kate, R. J., Wong, Y. W., and Mooney, R. J. (2005). Learning to transform natural to formal languages. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*.

Kehler, A. (2007). Rethinking the SMASH approach to pronoun interpretation. In *Interdisciplinary perspectives on reference processing*, New Directions in Cognitive Science Series, pages 95–122. Oxford University Press.

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.

Kim, M.-J. (2002). Does korean have adjectives? *MIT Working Papers in Linguistics*, 43:71–89.

Kim, S.-M. and Hovy, E. (2006). Extracting opinions, opinion holders, and topics expressed in online news media text. In *Proceedings of the Workshop on Sentiment and Subjectivity in Text*, pages 1–8, Sydney, Australia. Association for Computational Linguistics.

Kim, Y. (2014). Convolutional neural networks for sentence classification. In *Proceedings of Empirical Methods for Natural Language Processing (EMNLP)*, pages 1746–1751.

Kipper-Schuler, K. (2005). *VerbNet: A broad-coverage, comprehensive verb lexicon*. PhD thesis.

Klein, D. and Manning, C. D. (2003). Accurate unlexicalized parsing. In *Proceedings of the Association for Computational Linguistics (ACL)*, pages 423–430.

Klenner, M. (2007). Enforcing consistency on coreference sets. In *Recent Advances in Natural Language Processing (RANLP)*, pages 323–328.

Knight, K. (1999). Decoding complexity in word-replacement translation models. *Computational Linguistics*, 25(4):607–615.

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. (2005). Hidden-variable models for discriminative reranking. In *Proceedings of Empirical Methods for Natural Language Processing (EMNLP)*, pages 507–514.

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.

Krishnamurthy, J. (2016). Probabilistic models for learning a semantic parser lexicon. In *Proceedings of the North American Chapter of the Association for Computational Linguistics (NAACL)*, pages 606–616, San Diego, CA.

Krishnamurthy, J. and Mitchell, T. M. (2012). Weakly supervised training of semantic parsers. In *Proceedings of Empirical Methods for Natural Language Processing (EMNLP)*, pages 754–765.

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)*.

Kwiatkowski, T., Goldwater, S., Zettlemoyer, L., and Steedman, M. (2012). A probabilistic model of syntactic and semantic acquisition from child-directed utterances and their meanings. In *Proceedings of the European Chapter of the Association for Computational Linguistics (EACL)*, pages 234–244, Avignon, France.

Lafferty, J., McCallum, A., and Pereira, F. (2001). Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *icml*.

Lample, G., Ballesteros, M., Subramanian, S., Kawakami, K., and Dyer, C. (2016). Neural architectures for named entity recognition. In *Proceedings of the North American Chapter of the Association for Computational Linguistics (NAACL)*, pages 260–270, San Diego, CA.

Lappin, S. and Leass, H. J. (1994). An algorithm for pronominal anaphora resolution. *Computational linguistics*, 20(4):535–561.

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.

Law, E. and Ahn, L. v. (2011). Human computation. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 5(3):1–121.

LeCun, Y. and Bengio, Y. (1995). Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361.

Lee, C. M. and Narayanan, S. S. (2005). Toward detecting emotions in spoken dialogs. *IEEE transactions on speech and audio processing*, 13(2):293–303.

Lee, H., Chang, A., Peirsman, Y., Chambers, N., Surdeanu, M., and Jurafsky, D. (2013). Deterministic coreference resolution based on entity-centric, precision-ranked rules. *Computational Linguistics*, 39(4):885–916.

Lee, H., Peirsman, Y., Chang, A., Chambers, N., Surdeanu, M., and Jurafsky, D. (2011). Stanford's multi-pass sieve coreference resolution system at the conll-2011 shared task. In *Proceedings of the Conference on Natural Language Learning (CoNLL)*, pages 28–34. Association for Computational Linguistics.

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. (2014a). Dependency-based word embeddings. In *Proceedings of the Association for Computational Linguistics (ACL)*, pages 302–308, Baltimore, MD.

Levy, O. and Goldberg, Y. (2014b). Neural word embedding as implicit matrix factorization. In *Neural Information Processing Systems (NIPS)*, Montréal.

Li, J. and Jurafsky, D. (2015). Do multi-sense embeddings improve natural language understanding? In *Proceedings of Empirical Methods for Natural Language Processing (EMNLP)*, pages 1722–1732.

Liang, P., Bouchard-Côté, A., Klein, D., and Taskar, B. (2006). An end-to-end discriminative approach to machine translation. In *Proceedings of the Association for Computational Linguistics (ACL)*, pages 761–768.

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.

Liang, P. and Potts, C. (2015). Bringing machine learning and compositional semantics together. *Annual Review of Linguistics*, 1(1):355–376.

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., Dyer, C., Black, A., and Trancoso, I. (2015a). Two/too simple adaptations of word2vec for syntax problems. In *Proceedings of the North American Chapter of the Association for Computational Linguistics (NAACL)*, Denver, CO.

Ling, W., Luís, T., Marujo, L., Astudillo, R. F., Amir, S., Dyer, C., Black, A. W., and Trancoso, I. (2015b). 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.

Loveland, D. W. (2016). *Automated Theorem Proving: a logical basis*. Elsevier.

Ma, X. and Hovy, E. (2016). End-to-end sequence labeling via bi-directional lstm-cnns-crf. In *Proceedings of the Association for Computational Linguistics (ACL)*, pages 1064–1074, Berlin.

Manning, C. D. (2011). Part-of-speech tagging from 97% to 100%: is it time for some linguistics? In *International Conference on Intelligent Text Processing and Computational Linguistics*, pages 171–189. Springer.

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.

Martschat, S. and Strube, M. (2015). Latent structures for coreference resolution. *Transactions of the Association for Computational Linguistics*, 3:405–418.

Matthiessen, C. and Bateman, J. A. (1991). *Text generation and systemic-functional linguistics: experiences from English and Japanese*. Pinter Publishers.

May, J. (2016). Semeval-2016 task 8: Meaning representation parsing. In *Proceedings of SemEval*, pages 1063–1073.

McCallum, A. and Wellner, B. (2004). Conditional models of identity uncertainty with application to noun coreference. In *NIPS*, pages 905–912.

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.

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., Cernockŷ, 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. (1974). A framework for representing knowledge. Technical Report 306, MIT AI Laboratory.

Minsky, M. and Papert, S. (1969). *Perceptrons*. MIT press.

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.

Misra, D. K. and Artzi, Y. (2016). Neural shift-reduce ccg semantic parsing. In *Proceedings of Empirical Methods for Natural Language Processing (EMNLP)*.

Mnih, A. and Hinton, G. (2007). Three new graphical models for statistical language modelling. In *Proceedings of the 24th international conference on Machine learning*, ICML '07, pages 641–648, New York, NY, USA. ACM.

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)*.

Mohammad, S. M. and Turney, P. D. (2013). Crowdsourcing a word–emotion association lexicon. *Computational Intelligence*, 29(3):436–465.

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.

Montague, R. (1973). The proper treatment of quantification in ordinary english. In *Approaches to natural language*, pages 221–242. Springer.

Müller, C. and Strube, M. (2006). Multi-level annotation of linguistic data with mmax2. *Corpus technology and language pedagogy: New resources, new tools, new methods*, 3:197–214.

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.*

Neubig, G., Dyer, C., Goldberg, Y., Matthews, A., Ammar, W., Anastasopoulos, A., Ballesteros, M., Chiang, D., Clothiaux, D., Cohn, T., Duh, K., Faruqui, M., Gan, C., Garrette, D., Ji, Y., Kong, L., Kuncoro, A., Kumar, G., Malaviya, C., Michel, P., Oda, Y., Richardson, M., Saphra, N., Swayamdipta, S., and Yin, P. (2017). Dynet: The dynamic neural network toolkit.

Neuhaus, P. and Bröker, N. (1997). The complexity of recognition of linguistically adequate dependency grammars. In *eacl*, pages 337–343.

Nguyen, D. and Dogruöz23, A. S. (2013). Word level language identification in online multilingual communication. In *Proceedings of Empirical Methods for Natural Language Processing (EMNLP)*.

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. (2008). Algorithms for deterministic incremental dependency parsing. *Computational Linguistics*, 34(4):513–553.

Nivre, J., Agić, Ž., Ahrenberg, L., Aranzabe, M. J., Asahara, M., Atutxa, A., Ballesteros, M., Bauer, J., Bengoetxea, K., Berzak, Y., Bhat, R. A., Bick, E., Börstell, C., Bosco, C., Bouma, G., Bowman, S., Cebirolu Eryiit, G., Celano, G. G. A., Chalub, F., Çöltekin, Ç., Connor, M., Davidson, E., de Marneffe, M.-C., Diaz de Ilarraza, A., Dobrovoljc, K., Dozat, T., Droganova, K., Dwivedi, P., Eli, M., Erjavec, T., Farkas, R., Foster, J., Freitas, C., Gajdošová, K., Galbraith, D., Garcia, M., Gärdenfors, M., Garza, S., Ginter, F., Goenaga, I., Gojenola, K., Gökrmak, M., Goldberg, Y., Gómez Guinovart, X., Gonzáles Saavedra, B., Grioni, M., Grūzītis, N., Guillaume, B., Hajič, J., Hà M, L., Haug, D., Hladká, B., Ion, R., Irimia, E., Johannsen, A., Jørgensen, F., Kaşkara, H., Kanayama, H., Kanerva, J., Katz, B., Kenney, J., Kotsyba, N., Krek, S., Laippala, V., Lam, L., Lê Hng, P., Lenci, A., Ljubešić, N., Lyashevskaya, O., Lynn, T., Makazhanov, A., Manning, C., Mărănduc, C., Mareček, D., Martínez Alonso, H., Martins, A., Mašek, J., Matsumoto, Y., McDonald, R., Missilä, A., Mititelu, V., Miyao, Y., Montemagni, S., Mori, K. S., Mori, S., Moskalevskyi, B., Muischnek, K., Mustafina, N., Müürisep, K., Nguyn Th, L., Nguyn Th Minh, H., Nikolaev, V., Nurmi, H., Osenova, P., Östling, R., Øvrelid, L., Paiva, V., Pascual, E., Passarotti, M., Perez, C.-A., Petrov, S., Piitulainen, J., Plank, B., Popel, M., Pretkalnia, L., Prokopidis, P., Puolakainen, T., Pyysalo, S., Rademaker, A., Ramasamy, L., Real, L., Rituma, L., Rosa, R., Saleh, S., Saulīte, B., Schuster, S., Seeker, W., Seraji, M., Shakurova, L., Shen, M., Silveira, N., Simi, M., Simionescu, R., Simkó, K., Šimková, M., Simov, K., Smith, A., Spadine, C., Suhr, A., Sulubacak, U., Szántó, Z., Tanaka, T., Tsarfaty, R., Tyers, F., Uematsu, S., Uria, L., van Noord, G., Varga, V., Vincze, V., Wallin, L., Wang, J. X., Washington, J. N., Wirén, M., Žabokrtský, Z., Zeldes, A., Zeman, D., and Zhu, H. (2016). Universal dependencies 1.4. LINDAT/CLARIN digital library at the Institute of Formal and Applied Linguistics, Charles University in Prague.

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.

O'Connor, B., Krieger, M., and Ahn, D. (2010). Tweetmotif: Exploratory search and topic summarization for twitter. In *Proceedings of the International Conference on Web and Social Media (ICWSM)*, pages 384–385.

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.

Paice, C. D. (1990). Another stemmer. In *ACM SIGIR Forum*, volume 24, pages 56–61.

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.

Pan, X., Cassidy, T., Hermjakob, U., Ji, H., and Knight, K. (2015). Unsupervised entity linking with abstract meaning representation. In *Proceedings of the North American Chapter of the Association for Computational Linguistics (NAACL)*, pages 1130–1139, Denver, CO.

Pang, B. and Lee, L. (2004). A sentimental education: Sentiment analysis using subjectivity summarization based on minimum cuts. In *Proceedings of the Association for Computational Linguistics (ACL)*, pages 271–278.

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.

Parsons, T. (1990). *Events in the Semantics of English*, volume 5. MIT Press.

Pascanu, R., Mikolov, T., and Bengio, Y. (2013). On the difficulty of training recurrent neural networks. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 1310–1318.

Peng, F., Feng, F., and McCallum, A. (2004). Chinese segmentation and new word detection using conditional random fields. In *Proceedings of the International Conference on Computational Linguistics (COLING)*, page 562.

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. and Schabes, Y. (1992). Inside-outside reestimation from partially bracketed corpora. In *Proceedings of the Association for Computational Linguistics (ACL)*, pages 128–135.

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.

Pereira, F. C. N. and Shieber, S. M. (2002). *Prolog and natural-language analysis*. Microtome Publishing.

Peterson, W. W., Birdsall, T. G., and Fox, W. C. (1954). The theory of signal detectability. *Transactions of the IRE professional group on information theory*, 4(4):171–212.

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 McDonald, R. (2012). Overview of the 2012 shared task on parsing the web. In *Notes of the First Workshop on Syntactic Analysis of Non-Canonical Language (SANCL)*, volume 59.

Polanyi, L. and Zaenen, A. (2006). Contextual valence shifters. In *Computing attitude and affect in text: Theory and applications*. Springer.

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.

Popescu, A.-M., Etzioni, O., and Kautz, H. (2003). Towards a theory of natural language interfaces to databases. In *Proceedings of Intelligent User Interfaces (IUI)*, pages 149–157.

Porter, M. F. (1980). An algorithm for suffix stripping. *Program*, 14(3):130–137.

Pradhan, S., Ramshaw, L., Marcus, M., Palmer, M., Weischedel, R., and Xue, N. (2011). Conll-2011 shared task: Modeling unrestricted coreference in ontonotes. In *Proceedings of the Fifteenth Conference on Computational Natural Language Learning: Shared Task*, pages 1–27. Association for Computational Linguistics.

Pradhan, S., Ward, W., Hacioglu, K., Martin, J. H., and Jurafsky, D. (2005). Semantic role labeling using different syntactic views. In *Proceedings of the Association for Computational Linguistics (ACL)*, pages 581–588, Ann Arbor, Michigan.

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.

Qiu, G., Liu, B., Bu, J., and Chen, C. (2011). Opinion word expansion and target extraction through double propagation. *Computational linguistics*, 37(1):9–27.

Quattoni, A., Wang, S., Morency, L.-P., Collins, M., and Darrell, T. (2007). Hidden conditional random fields. *IEEE transactions on pattern analysis and machine intelligence*, 29(10).

Rabiner, L. (1989). A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286.

Rao, D., Yarowsky, D., Shreevats, A., and Gupta, M. (2010). Classifying latent user attributes in twitter. In *Proceedings of Workshop on Search and mining user-generated contents*.

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. (1996). A maximum entropy model for part-of-speech tagging. In *emnlp*, pages 133–142.

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, D., Rudinger, R., Ferraro, F., Harman, C., Rawlins, K., and Durme, B. V. (2015). Semantic proto-roles. *Transactions of the Association for Computational Linguistics*, 3:475–488.

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.

Ribeiro, F. N., Araújo, M., Gonçalves, P., Gonçalves, M. A., and Benevenuto, F. (2016). Sentibench-a benchmark comparison of state-of-the-practice sentiment analysis methods. *EPJ Data Science*, 5(1):1–29.

Riloff, E. and Wiebe, J. (2003). Learning extraction patterns for subjective expressions. In *Proceedings of the 2003 conference on Empirical methods in natural language processing*, pages 105–112. Association for Computational Linguistics.

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.

Rush, A. M., Sontag, D., Collins, M., and Jaakkola, T. (2010). On dual decomposition and linear programming relaxations for natural language processing. In *Proceedings of Empirical Methods for Natural Language Processing (EMNLP)*, pages 1–11.

Salinas, E. and Abbott, L. (1996). A model of multiplicative neural responses in parietal cortex. *Proceedings of the national academy of sciences*, 93(21):11956–11961.

Santorini, B. (1990). Part-of-speech tagging guidelines for the penn treebank project (3rd revision). Technical Report MS-CIS-90-47, University of Pennsylvania.

Sato, M.-A. and Ishii, S. (2000). On-line em algorithm for the normalized gaussian network. *Neural computation*, 12(2):407–432.

Schank, R. C. and Abelson, R. (1977). *Scripts, goals, plans, and understanding*. Erlbaum, Hillsdale, NJ.

Schneider, N., Flanigan, J., and O'Gorman, T. (2015). The logic of amr: Practical, unified, graph-based sentence semantics for nlp. In *Proceedings of the North American Chapter of the Association for Computational Linguistics (NAACL)*, pages 4–5, Denver, CO.

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.

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.

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)*.

Solorio, T. and Liu, Y. (2008). Learning to predict code-switching points. In *Proceedings of Empirical Methods for Natural Language Processing (EMNLP)*, pages 973–981, Honolulu, HI. Association for Computational Linguistics.

Somasundaran, S. and Wiebe, J. (2009). Recognizing stances in online debates. In *Proceedings of the Association for Computational Linguistics (ACL)*, pages 226–234, Suntec, Singapore.

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.

Sowa, J. F. (2000). *Knowledge representation: logical, philosophical, and computational foundations*. Brooks/Cole, Pacific Grove, CA.

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.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958.

Stenetorp, P., Pyysalo, S., Topić, G., Ohta, T., Ananiadou, S., and Tsujii, J. (2012). Brat: a web-based tool for nlp-assisted text annotation. In *Proceedings of the European Chapter of the Association for Computational Linguistics (EACL)*, pages 102–107, Avignon, France.

Stolcke, A., Ries, K., Coccaro, N., Shriberg, E., Bates, R., Jurafsky, D., Taylor, P., Martin, R., Van Ess-Dykema, C., and Meteer, M. (2000). Dialogue act modeling for automatic tagging and recognition of conversational speech. *Computational linguistics*, 26(3):339–373.

Stone, P. J. (1966). *The General Inquirer: A Computer Approach to Content Analysis*. The MIT Press.

Stoyanov, V., Gilbert, N., Cardie, C., and Riloff, E. (2009). Conundrums in noun phrase coreference resolution: Making sense of the state-of-the-art. In *Proceedings of the Association for Computational Linguistics (ACL)*, pages 656–664, Suntec, Singapore.

Sun, X., Matsuzaki, T., Okanohara, D., and Tsujii, J. (2009). Latent variable perceptron algorithm for structured classification. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, volume 9, pages 1236–1242.

Sundermeyer, M., Schlüter, R., and Ney, H. (2012). Lstm neural networks for language modeling. In *INTERSPEECH*.

Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *Neural Information Processing Systems (NIPS)*, pages 3104–3112, Montréal.

Taboada, M., Brooke, J., Tofiloski, M., Voll, K., and Stede, M. (2011). Lexicon-based methods for sentiment analysis. *Computational linguistics*, 37(2):267–307.

Täckström, O., Ganchev, K., and Das, D. (2015). Efficient inference and structured learning for semantic role labeling. *Transactions of the Association for Computational Linguistics*, 3:29–41.

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.

Toutanova, K., Klein, D., Manning, C. D., and Singer, Y. (2003). Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of NAACL*.

Tromble, R. W. and Eisner, J. (2006). A fast finite-state relaxation method for enforcing global constraints on sequence decoding. In *Proceedings of the North American Chapter of the Association for Computational Linguistics (NAACL)*, page 423, New York, NY.

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.

Turian, J., Ratinov, L., and Bengio, Y. (2010). Word representations: A simple and general method for semi-supervised learning. In *Proceedings of the Association for Computational Linguistics (ACL)*, pages 384–394, Uppsala, Sweden.

Twain, M. (1997). *A Tramp Abroad*. Penguin, New York.

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.

Wang, C., Xue, N., and Pradhan, S. (2015). A Transition-based Algorithm for AMR Parsing. In *Proceedings of the North American Chapter of the Association for Computational Linguistics (NAACL)*, pages 366–375, Denver, CO.

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.

Weinberger, K., Dasgupta, A., Langford, J., Smola, A., and Attenberg, J. (2009). Feature hashing for large scale multitask learning. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 1113–1120.

Wiebe, J., Wilson, T., and Cardie, C. (2005). Annotating expressions of opinions and emotions in language. *Language resources and evaluation*, 39(2):165–210.

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.

Winograd, T. (1972). Understanding natural language. *Cognitive psychology*, 3(1):1–191.

Wong, Y. W. and Mooney, R. J. (2006). Learning for semantic parsing with statistical machine translation. In *Proceedings of the North American Chapter of the Association for Computational Linguistics (NAACL)*, pages 439–446, New York, NY.

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.

Xue, N. et al. (2003). Chinese word segmentation as character tagging. *Computational Linguistics and Chinese Language Processing*, 8(1):29–48.

Yamada, H. and Matsumoto, Y. (2003). Statistical dependency analysis with support vector machines. In *Proceedings of IWPT*, volume 3, pages 195–206.

Yang, Y. and Eisenstein, J. (2013). A log-linear model for unsupervised text normalization. In *Proceedings of Empirical Methods for Natural Language Processing (EMNLP)*.

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.

Yu, C.-N. J. and Joachims, T. (2009). Learning structural svms with latent variables. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 1169–1176.

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.

Zelle, J. M. and Mooney, R. J. (1996). Learning to parse database queries using inductive logic programming. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 1050–1055.

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. and Clark, S. (2008). A tale of two parsers: investigating and combining graph-based and transition-based dependency parsing using beam-search. In *Proceedings of Empirical Methods for Natural Language Processing (EMNLP)*, pages 562–571, Honolulu, HI.

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.

Zhang, Y. and Nivre, J. (2011). Transition-based dependency parsing with rich non-local features. In *Proceedings of the Association for Computational Linguistics (ACL)*, pages 188–193, Portland, OR.

Zhou, J. and Xu, W. (2015). End-to-end learning of semantic role labeling using recurrent neural networks. In *Proceedings of the Association for Computational Linguistics (ACL)*, pages 1127–1137, Beijing.

Index

F-measure, 66
K-means, 78
α-conversion, 245
β-conversion, 243
β-reduction, 243
k-nearest-neighbors, 50

ablation tests, 67
Abstract Meaning Representation (AMR), 257, 271
access (an entity), 298
accuracy, 64
activation function, 103
AdaGrad, 45
adequacy (translation), 320
adjectives, 146
adjuncts (semantics), 256, 260
adverbs, 146
agent (thematic role), 257
alignment, 273, 319, 322
Amazon Mechanical Turk, 74
anaphora resolution, 297
antecedent (coreference), 305
antecedent structure, 305
antonymy, 58, 281
Arc-eager dependency parsing, 229
arc-factored, 221
arc-standard, 227
area under the curve (AUC), 67
Attachment ambiguity, 194
attention vector, 330
attributive adjectives, 146
automated theorem provers, 238
auxiliary verbs, 146

backpropagation, 50, 104
backpropagation through time, 106
balanced test set, 65
Baum-Welch algorithm, 139
Bayes' rule, 16
Beam search, 230
bias, 95
bias-variance, 97
bidirectional recurrent neural network, 138
Bigrams, 31
bigrams, 54
bilexical, 208
Bilexical features, 221
binarize, 193
binding (anaphora), 299
binomial distribution, 68
binomial random variable, 20
binomial test, 68
BIO notation, 152, 268
bipartite graph, 274
bitext, 94, 319, 324
Bonferroni correction, 71
boosting, 49
bootstrap samples, 69
bound variable, 240
Brown clusters, 283

c-command, 299
Catalan number, 191
cataphora, 298
centering theory, 298, 301
chain rule, 16
chance agreement, 73
character-level language models, 111
CKY, 191
class imbalance, 64
cleft, 302
closed class, 145
closed-vocabulary, 110
Cohen's Kappa, 73
collapsed dependency grammar, 218
collocation features, 59
common nouns, 144
comparative adverbs, 146
complement event (probability), 14
Complement structure, 195
computation graph, 106
concept (AMR), 271
conditional independence, 18
conditional probability, 16
conditional probability distribution, 19
Conditional Random Field, 133
conditional recurrent neural network, 329
conditionally independent, 122
confidence interval, 70
connected (graph theory), 273
consistency (logic), 241
constants (logic), 236
constrained optimization, 265
constraints, 264
convexity, 35, 250
convolutional neural network, 111, 330
convolutional neural networks, 54, 268
coordinating conjunctions, 147
Coordination scope, 195
copula, 146
Coreference resolution, 297
cost-augmented decoding, 39, 132
count, 64
count nouns, 144
crowdsourcing, 74
cumulative probability distribution, 69

decidability (logic), 241
decision stumps, 49
decision trees, 49
deep learning, 50
degree (of events), 146
denotation (semantics), 236
dependency grammar, 216
dependency graph, 216
dependency path, 59, 264
dependent, 216
derivation (semantic parsing), 243, 248
determiners, 147
development set, 34, 64
dialogue acts, 73
digital humanities, 53
discount, 98
discrete random variable, 18
disjoint events, 15
distributional, 282
distributional hypothesis, 281, 282
distributional statistics, 59
dropout, 106
dual decomposition, 268
dual form, 50
dynamic programming, 117
dynamic semantics, 253

early stopping, 34
effective counts, 98
emotion, 55
encoder-decoder architecture, 329
ensemble, 270
entailment, 241
entity (coreference), 305
Entity linking, 297

entity linking, 316
entropy, 48
estimation, 20
event (probability), 14
event semantics, 255
evoke (an entity), 298
existential quantifier, 240
existential there, 148
expectation, 19
expectation maximization, 79, 99
explicit semantic analysis, 284
extraposition, 302

factoid questions, 274
false discovery rate, 71
False negative, 65
False positive, 65
false positive, 17
false positive rate, 66
feasible set, 265
feature function, 31
feature hashing, 64
feature templates, 130
first-order logic, 238
fluent, 93
focus, 272
forward algorithm, 202
forward variables, 126
forward-backward algorithm, 135
frame elements, 261
FrameNet, 261
frames, 261
free (anaphora), 299
free variable, 239
function (first-order logic), 240

garden path sentence, 114
gazetteers, 304
gazzeteers, 153, 318
Generalization, 34
generative model, 22
generative process, 99
Gibbs Sampling, 141
gloss, 94
gradable adjectives, 146
gradient descent, 38
gradient-based optimization, 43
graph-based dependency parsing, 220
graphical model, 122

Hamming cost, 131
hanzi, 61
hard EM, 83
hard tanh, 269
head, 216
head word, 303
head words, 215
held-out data, 108
hidden Markov models, 122
hierarchical softmax, 104
hinge loss, 36
human computation, 74
hyponymy, 58, 281

Indicator random variables, 18
inference, 115
inference (logic), 235
inference rules, 238
infinitive, 147
inflectional affixes, 62
information provenance, 316
information status, 147
input word embeddings, 103
inside algorithm, 202
inside-outside algorithm, 211
instance (AMR), 271
instance labels, 13
Integer linear programming, 307
integer linear programming, 265
inter-annotator agreement, 73
interpolation, 99
inverse relation (AMR), 272

irrealis, 54

Jeffreys-Perks law, 97
joint probabilities, 19

Kalman Smoother, 140
Katz backoff, 99
kernel methods, 50
knowledge base population, 316

labeled dependencies, 217
lambda calculus, 242
lambda expressions, 243
language model, 94
Laplace smoothing, 97
large-margin, 36
latent conditional random fields, 251
latent semantic analysis, 284
latent variable, 319
latent variable perceptron, 251
latent variables, 249
least squares, 56
lemma, 57
lemmatization, 62
lexical entry, 244
lexical features (linguistics), 149
lexical semantics, 57
lexical unit (frame semantics), 261
lexicalized tree-adjoining grammar, 208
lexicon, 244
lexicon-based classification, 56
lexicon-based sentiment analysis, 54
Lidstone smoothing, 97
light verb, 146, 273
likelihood, 16, 40
linear regression, 56
linear separability, 32
Logistic regression, 39
Long short-term memories, 104
long short-term memories, 270
long short-term memory, 107, 329
loss function, 35
LSTM, 107
LSTM-CRF, 139, 270

Macro F-measure, 66
macro-reading, 316
manner (of events), 146
margin, 33, 36
margin loss, 37
marginal probability distribution, 19
marginalize, 15
markable, 305
Markov blanket, 122
Markov Chain Monte Carlo (MCMC), 141
mass nouns, 144
matrix-tree theorem, 223
max pooling, 268
max-margin Markov network, 131
max-product algorithm, 125
maximum a posteriori, 21
maximum conditional likelihood, 47
maximum entropy, 47
maximum likelihood, 20
maximum likelihood estimate, 25
McNemar's test, 68
meaning representation, 235
mention (coreference), 305
mention ranking, 306
mention-pair model, 305
meronymy, 58, 281
micro F-measure, 66
micro-reading, 316
mini-batch, 45
modals, 145
model builder, 241
model checker, 241
model-theoretic semantics, 236
Modifier scope, 195
modus ponens, 238
morphemes, 111
Morphology, 129

morphology, 62
multi-task learning, 269
multinomial distribution, 22

n-grams, 31
Naïve Bayes, 21
named entity recognition, 139, 316
named entity types, 316
negation, 54
Neo-Davidsonian event semantics, 256
neural attention, 330
neural machine translation, 320
neural networks, 102
noise-contrastive estimation, 104
noisy channel model, 94, 321
nominals, 297
nominals (coreference), 304
non-core roles (AMR), 272
non-projective dependency parsing, 223
norm, 37
normalization, 62
nouns, 144
null hypothesis, 67

one-tailed p-value, 69
online learning, 32
open word classes, 144
opinion polarity, 53
oracle, 274
overfitting, 34
overgenerate, 247

parameters, 20
parent annotation, 204
part-of-speech, 143
part-of-speech tagging, 113
Particle versus preposition, 195
particles (part-of-speech), 147
partition, 15
partition function, 136
Passive-Aggressive, 46
Penn Treebank, 201
perplexity, 109
personal pronouns, 144
pleonastic, 302
polysemous, 58
pooling (in convolutional neural networks), 269
possessive pronouns, 145
posterior, 16
potentials, 135
pre-determiners, 148
pre-trained word embeddings, 269
precision, 65
predicative adjective, 145
predicative adjectives, 146
prepositions, 147
presence, 64
principle of compositionality, 242
prior, 16
prior expectation, 21
probabilistic context-free grammar, 198
probabilistic models, 20
probability distribution, 19
probability mass function, 68
projectivity, 219
pronouns, 144
PropBank, 260
proper nouns, 144
property (logic), 239
propositions, 236, 237
proto-roles, 259

random outcomes, 14
ranking, 56
recall, 65
receiver operating characteristic (ROC), 66
rectified linear unit, 103
rectified linear unit (ReLU), 269
recurrent neural network, 102, 270
reference arguments, 277
regression, 56

reification (events), 255
relation extraction, 275
relations (logic), 236
relative frequency estimate, 21
relative frequency estimator, 25
ReLU, 103
reranking, 212
ridge regression, 56

sample space, 14
satisfaction (logic), 241
second-order logic, 239
seed lexicon, 57
semantic concordance, 59
semantic parsing, 242
semantic role, 256
Semantic role labeling, 256
semantic underspecification, 252
semantics, 203
semiring algebra, 127
semisupervised, 60
senses, 259
sentence (logic), 240
sentiment, 53
shift-reduce, 191
sigmoid function, 103
slack variables, 37
smoothing, 97
soft K-means, 78
softmax, 47, 102
source language, 319
sparsity, 47
spectral learning, 83
split constituents, 262
Spurious ambiguity, 228
spurious ambiguity, 248
stand-off annotations, 73
statistical learning theory, 33
statistical machine translation, 319
statistical significance, 68
stemmer, 62
Stochastic gradient descent, 44
stoplist, 63
stopwords, 63
structured perceptron, 130
structured support vector machine, 131
subcategorization, 181
subgradient, 47
subgradient set, 38
subjectivity detection, 54
subordinating conjunctions, 147
sum-product, 127
supervised machine learning, 13
Support Vector Machine, 37
synonymy, 58, 281
synsets, 58
syntactic dependency, 215
syntactic path, 263
syntax, 143

target language, 319
Targeted sentiment analysis, 55
test statistic, 68
thematic roles, 257
third axiom of probability, 15
token, 93
tokenization, 60
Tokenizers, 93
training data, 25
translation model, 94
Tree-Adjoining Grammar, 201
trellis, 118
tropical semiring, 128
True negative, 65
True positive, 65
true positive, 17
true positive rate, 66
truth conditions, 240
tuning set, 64
two-tailed test, 69
type systems, 246
type-raising, 245

uniqueness, 147
universal quantifier, 240
unsupervised, 60
unsupervised learning, 55

validation function (semantic parsing), 252
validity (logic), 241
variable (AMR), 271
variance, 70
VC-dimension, 49
VerbNet, 258
verbs, 145
vertical Markovization, 204
Viterbi algorithm, 117
Viterbi variable, 117

weighted context-free grammar, 198
wh-determienrs, 148
wh-pronouns, 145
word embeddings, 104, 282, 283
word representations, 282
word sense disambiguation, 57
word senses, 57
word tokens, 60
world model, 236

zero-one loss, 35