Linear Models for Statistical Natural Language Processing

Jacob Eisenstein

September 9, 2014

Chapter 1

Introduction

This is a collection of notes that I use for teaching Georgia Tech Computer Science 4650 and 7650, "Natural Language." The notes focus on what I view as a core subset of the field of natural language processing, unified by the concept of linear models. This includes approaches to 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. The theme was inspired by Fernando Pereira's EMNLP 2008 keynote, "Are linear models right for language."[1] The notes are heavily influenced by several other good resources (e.g., Manning and Schütze, 1999; Jurafsky and Martin, 2009; Figueiredo et al., 2013; Collins, 2013), but for various reasons I wanted to create something of my own.

[1] You can see a version of this talk — not the one I saw — online at `vimeo.com/30676245`

Chapter 2

Notation

w_n	word token at position n
$\boldsymbol{x}_i$	a vector of feature counts for instance i, often word counts
N	number of training instances
V	number of words in vocabulary
$\boldsymbol{\theta}$	a vector of weights
y_i	the label for instance i
$\boldsymbol{y}$	vector of labels across all instances
$\mathcal{Y}$	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
$P(A)$	probability function of event A
$\mathrm{p}_B(b)$	the marginal probability of random variable B taking value b

Chapter 3

Linear classification and features

Suppose you want to build a spam detector. Spam vs. 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.

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 *c1al1s*? How about *Bayesian*? One approach would be to define a "spamminess" score for every word in the dictionary, and then just add them up. This is also a commonly-used approach to sentiment analysis, where each word is scored as one of $\{1, 0, -1\}$, with 1 indicating positive sentiment and -1 indicating negative sentiment.

These scores are called **weights**, written $\boldsymbol{\theta}$, and we'll spend a lot of time later 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 is an element y_i in a set of K possible labels $\mathcal{Y}$. Then for any pair $\langle \boldsymbol{x}_i, y_i \rangle$, we can define a *feature vector* $\boldsymbol{f}(\boldsymbol{x}_i, y_i)$, such that:

$$\boldsymbol{f}(\boldsymbol{x}, y = 0) = [\boldsymbol{x}_i^\top\ \mathbf{0}_{V(K-1)}^\top]^\top \tag{3.1}$$

$$\boldsymbol{f}(\boldsymbol{x}, y = 1) = [\mathbf{0}_V^\top\ \boldsymbol{x}_i^\top\ \mathbf{0}_{V(K-2)}^\top]^\top \tag{3.2}$$

$$\boldsymbol{f}(\boldsymbol{x}, y = 2) = [\mathbf{0}_{2V}^\top\ \boldsymbol{x}_i^\top\ \mathbf{0}_{V(K-3)}^\top]^\top \tag{3.3}$$

$$\ldots \tag{3.4}$$

$$\boldsymbol{f}(\boldsymbol{x}, K) = [\mathbf{0}_{V(K-1)}^\top\ \boldsymbol{x}_i^\top]^\top, \tag{3.5}$$

where $\mathbf{0}_{VK}$ is a column vector of VK zeros. 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 $(V+1)K$.

Now, given a vector of weights, $\boldsymbol{\theta} \in \mathcal{R}^{(V+1)K}$, we can compute the inner product $\boldsymbol{\theta}^\top \boldsymbol{f}(x, y)$. Then for any document $\boldsymbol{x}_i$, we can predict a label $\hat{y}$ as

$$\hat{y} = \arg\max_y \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}_i, y) \tag{3.6}$$

We could just set the weights by hand. If we wanted to distinguish, say, English from Spanish, we could just use English and Spanish dictionaries, and set each weight to 1. For example,

$$\begin{aligned} \theta_{\text{english,bicycle}} =& 1 & \qquad \theta_{\text{spanish,bicycle}} =& 0 \\ \theta_{\text{english,bicicleta}} =& 0 & \theta_{\text{spanish,bicicleta}} =& 1 \\ \theta_{\text{english,con}} =& 1 & \theta_{\text{spanish,con}} =& 1 \\ \theta_{\text{english,ordinateur}} =& 0 & \theta_{\text{spanish,ordinateur}} =& 0 \end{aligned}$$

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 Project 1.

But it's usually not easy to set the weights by hand. Instead, we will learn them from data. For example, suppose that an email user has manually labeled thousands of messages as "spam" or "not spam"; or a newspaper may label its 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.

3.1 Review of basic probability

This section is inspired/borrowed from Manning and Schütze (1999).

- **Formally**: When we write $P(\dot{})$, this denotes a function $P : \mathcal{F} \rightarrow [0, 1]$ from an **event space** $\mathcal{F}$ to a **probability**. A probability is a real number between zero and one, with zero representing impossibility and one representing certainty.

- The probabilities of disjoint event sets are additive: $A_i \cap A_j = \varnothing \Rightarrow P(A_i \cup A_j) = P(A_i) + P(A_j)$. This is a restatement of the Third Axiom of probability.

- For example, you might ask what is the probability of two heads on three coin flips. There are eight possible series of three flips $HHH, HHT, \ldots$, and each is an equally likely event. Of these events, three meet the criterion, HHT, HTH, THH. So the probability is $\frac{3}{8}$.

- More generally, $P(A_i \cup A_j) = P(A_i) + P(A_j) - P(A_i \cap A_j)$. This can be derived from the third axiom.

$$P(A_i \cup A_j) = P(A_i) + P(A_j - (A_i \cap A_j)) \tag{3.7}$$
$$P(A_j) = P(A_j - (A_i \cap A_j)) + P(A_i \cap A_j) \tag{3.8}$$
$$P(A_j - (A_i \cap A_j)) = P(A_j) - P(A_i \cap A_j) \tag{3.9}$$
$$P(A_i \cup A_j) = P(A_i) + P(A_j) - P(A_i \cap A_j) \tag{3.10}$$

- If the probability $P(A \cap B) = P(A)P(B)$, then the events A and B are *independent*, written $A \perp B$.

Conditional probability and Bayes' Rule

A conditional probability is an expression like $P(A \mid B)$, where we are interested in the probability of A conditioned on B happening.

- Conditional probability: $P(A \mid B) = P(A \cap B)/P(B)$

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

- Chain rule: $P(A \cap B) = P(A \mid B)P(B)$, which is just a rearrangement of terms.

- We can apply the chain rule multiple times:

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

  We'll do this a lot later in the course.

- Bayes' rule follows from the Chain rule: $P(A \mid B) = P(A \cap B)/P(B) = P(B \mid A)P(A)/P(B)$

Often we want the maximum a posteriori (MAP) estimate

$$\begin{aligned}\hat{B} &= \arg\max_B P(B \mid A)\\ &= \arg\max_B P(A \mid B)P(B)/P(A)\\ &\propto \arg\max_B P(A \mid B)P(B)\end{aligned}$$

- We don't need to normalize the probability because $P(A)$ is the same for all values of B.
- If we do need to compute the conditional $P(A \mid B)$, we can compute $P(A)$ by summing over $P(A \cap B) + P(A \cap \overline{B})$, where $B \cap \overline{B} = \varnothing$ and $B \cup \overline{B} = \Omega$, the entire sample space (such that $P(\Omega) = 1$).
- More generally, if $\bigcup_i B_i = \Omega$ and $\forall_{i,j}, B_i \cap B_j = \varnothing$, then $P(A) = \sum_i P(A \mid B_i)P(B_i)$.

Example Manning and Schütze (1999) have a nice example of Bayes Rule (Bayes Law) in a linguistic setting.

- Suppose one is interested in a rare syntactic construction, perhaps parasitic gaps, which occurs on average once in 100,000 sentences.
  - (An example of a sentence with a parasitic gap is *Which class did you attend __ without registering for __?* -JE)
- Lana Linguist has developed a complicated pattern matcher that attempts to identify sentences with parasitic gaps. Its pretty good, but it's not perfect:
  - If a sentence has a parasitic gap, it will say so with probability 0.95 (this is the **recall** -JE).
  - If it doesn't, it will wrongly say it does with probability 0.005 (this is the **false positive rate**, the additive inverse of **precision** -JE).
- Suppose the test says that a sentence contains a parasitic gap. What is the probability that this is true?
- (This example is usually framed in terms of tests for rare diseases. -JE)

Solution: Let G be the event of a sentence having a parasitic gap, and T be the event of the test being positive.

$$P(G \mid T) = \frac{P(G \mid T)P(T)}{P(G \mid T)P(T) + P(G \mid \overline{T})P(\overline{T})} \tag{3.11}$$

$$= \frac{0.95 \times 0.00001}{0.95 \times 0.00001 + 0.005 \times 0.99999} \approx 0.002 \tag{3.12}$$

Random variables

A random variable takes on a specific value in $\mathbb{R}^n$, typically with $n = 1$, but not always. Discrete random variables can take values only in some countable subset of $\mathbb{R}$.

- Recall the coin flip example. The number of heads, H, can be viewed as a discrete random variable, $H \in 0, 1, 2, 3$.
- The probability mass associated with each number is $\{\frac{1}{8}, \frac{3}{8}, \frac{3}{8}, \frac{1}{8}\}$.
- This set of numbers represents the **probability distribution** over H, written $P(H = h) = \mathrm{p}(h)$.
- To indicate that the RV H is distributed as $\mathrm{p}(h)$, we write $H \sim \mathrm{p}(h)$.
- The function $\mathrm{p}(h)$ is called a probability **mass** function (pmf) if h is discrete, and a probability **density** function (pdf) if h is continuous.
- If we have more than one variable, we can write a joint probability $\mathrm{p}(a, b) = P(A = a, B = b)$.
- We can write a **marginal** probability $\mathrm{p}_A(a) = \sum_b \mathrm{p}(a, b)$.
- Random variables are independent iff $\mathrm{p}_{A,B}(a, b) = \mathrm{p}_A(a)\mathrm{p}_B(b)$.
- We can write a conditional probability as $\mathrm{p}(a \mid b) = \frac{\mathrm{p}(a,b)}{\mathrm{p}_B(b)}$.

Expectations

Sometimes we want the **expectation** of a function, such as $E[g(x)] = \sum_{x \in \mathcal{X}} g(x)\mathrm{p}(x)$.

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:

$$E[g(x)] = \int_{\mathcal{X}} g(x)\mathrm{p}(x)dx \tag{3.13}$$

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

3.2 Naïve Bayes

Back to classification! A Naïve Bayes classifier chooses the weights $\boldsymbol{\theta}$ to maximize the *joint* probability of a labeled dataset, $\mathrm{p}(\boldsymbol{x}_{1:N}, \boldsymbol{y}_{1:N})$, where $\langle \boldsymbol{x}_i, y_i \rangle$ is a labeled instance.

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

- For each document i,
  - draw the label $y_i \sim$ Categorical(μ)
  - draw the vector of counts $\boldsymbol{x}_i \sim$ Multinomial(ϕ_{y_i}),

[1] 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.

The first thing this generative model tells us is that we can 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).

$$\mathrm{p}(\boldsymbol{x}, \boldsymbol{y}; \mu, \phi) = \prod_i \mathrm{p}(\boldsymbol{x}_i, y_i; \mu, \phi) \tag{3.14}$$

This means that the words in each document are **conditionally independent** given the parameters μ and ϕ.

When we write $y_i \sim \text{Categorical}(\mu)$, that means y_i is a stochastic draw from a categorical distribution with **parameter** μ. A categorical distribution is just like a weighted die: $\mathrm{p}_{\text{cat}}(y; \mu) = \mu_y$, where μ_y is the probability of the outcome $Y = y$. We require $\sum_y \mu_y = 1$ and $\forall_y, \mu_y \geq 0$.

A multinomial distribution is only slightly more complex:

$$\mathrm{p}_{\text{mult}}(\boldsymbol{x}; \phi) = \frac{\left(\sum_j x_j\right)!}{\prod_j x_j!} \prod_j \phi_j^{x_j} \tag{3.15}$$

We again require that $\sum_j \phi_j = 1$ and $\forall j, \phi_j \geq 0$. The first part of the equation doesn't depend on ϕ, and can usually be ignored. Can you see why we need the first part at all?[2]

We can write $\mathrm{p}(\boldsymbol{x}_i \mid y_i; \phi)$ to indicate the conditional probability of word counts $\boldsymbol{x}_i$ given label y_i, with parameter ϕ, which is equal to $\mathrm{p}_{\text{mult}}(\boldsymbol{x}_i; \phi_{y_i})$.

By specifying the multinomial distribution, we are working with *multinomial naïve Bayes* (MNB). Why "naïve"? Because the multinomial distribution treats each word token independently: the probability mass function factorizes across the counts.[3] We'll see this more clearly later, when we show how MNB is an example of linear classification.

Another version of Naïve Bayes

Consider a slight modification to the generative story of NB:

[2]Technically, a multinomial distribution requires a second parameter, the total number of counts (the number of words in the document). Even more technically, that number should be treated as a random variable, and drawn from some other distribution. But none of that matters for classification.

[3]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 generated independently.

- For each document i
  - Draw the label $y_i \sim \text{Categorical}(\mu)$
  - For each word $n \leq D_i$
    - Draw the word $w_{i,n} \sim \text{Categorical}(\phi_{y_i})$

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

However, as a classifier, it is identical to MNB. The final probabilities are reduced by a factor corresponding to the normalization term in the multinomial, $\frac{(\sum_j x_j)!}{\prod_j x_j!}$. This means that the resulting probabilities for a given $\boldsymbol{x}$ are different. However, none of this has anything to do with the label y or the parameters $\boldsymbol{\phi}$. The ratio of probabilities between any two labels y_1 and y_2 will be identical, as will the maximum likelihood estimates for the parameters μ and $\boldsymbol{\phi}$ (defined later).

Prediction

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

$$\begin{aligned}
\hat{y} =& \arg\max_y \text{p}(\boldsymbol{x}, y; \mu, \phi) \\
=& \arg\max_y \text{p}(\boldsymbol{x} \mid y; \phi)\text{p}(y; \mu) \\
=& \arg\max_y \log \text{p}(\boldsymbol{x} \mid y; \phi) + \log \text{p}(y; \mu)
\end{aligned}$$

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

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

$$\begin{aligned}
\log \mathrm{p}(\boldsymbol{x}, y; \mu, \phi) =& \arg\max_y \log \mathrm{p}(\boldsymbol{x} \mid y; \phi) + \log \mathrm{p}(y; \mu) \\
=& \log \left[\frac{\left(\sum_j x_j\right)!}{\prod_j x_j!} \prod_j \phi_{y,j}^{x_j} \right] + \log \mu_y \\
=& \log \frac{\left(\sum_j x_j\right)!}{\prod_j x_j!} + \sum_j x_j \log \phi_{y,j} + \log \mu_y \\
\propto& \sum_j x_j \log \phi_{y,j} + \log \mu_y \\
=& \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}, y),
\end{aligned}$$

where

$$\begin{aligned}
\boldsymbol{\theta} &= [\boldsymbol{\theta}^{(1)\top}, \boldsymbol{\theta}^{(2)\top}, \ldots, \boldsymbol{\theta}^{(K)\top}]^\top \\
\boldsymbol{\theta}^{(y)} &= [\log \phi_{y,1} \;\; \log \phi_{y,2} \;\; \ldots \;\; \log \phi_{y,M} \; \log \mu_y]^\top
\end{aligned}$$

and $\boldsymbol{f}(\boldsymbol{x}, y)$ is a vector of word counts and an offset, padded by zeros for the labels not equal to y (see equations 3.1-3.5). This ensures that the inner product $\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}, y)$ only activates the features in $\boldsymbol{\theta}^{(y)}$, which are what we need to compute the joint log-probability $\log \mathrm{p}(\boldsymbol{x}, y)$ for each y.

Estimation

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

$$\phi_{y,j} = \frac{\sum_{i:Y_i=y} x_{i,j}}{\sum_{j'} \sum_{i:Y_i=y} x_{i,j'}} = \frac{\text{count}(y, j)}{\sum_{j'} \text{count}(y, j')} \tag{3.16}$$

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

As in prediction, we want to maximize the joint likelihood of the data,

$$L = \sum_i \log \mathrm{p}_{\text{mult}}(\boldsymbol{x}_i; \boldsymbol{\phi}_{y_i}) + \log \mathrm{p}_{\text{cat}}(y_i; \mu) \tag{3.17}$$

Since $\mathrm{p}(y)$ is unrelated to ϕ, we can forget about it for now. But before we can just optimize L, we have to deal with a constraint:

$$\sum_j \phi_{y,j} = 1 \tag{3.18}$$

We'll do this by adding a Lagrange multiplier. Here's the resulting Lagrangian:

$$\ell[\boldsymbol{\phi}_y] = \sum_{i:Y_i=y} \sum_j x_{ij} \log \phi_{y,j} + \lambda(\sum_j \phi_{y,j} - 1) \tag{3.19}$$

We solve by setting $\frac{\partial \ell}{\boldsymbol{\phi}_j} = 0$.

$$\begin{aligned}
0 =& \sum_{i:Y_i=y} x_{i,j}/\phi_{y,j} - \lambda \\
\lambda\phi_{y,j} =& \sum_{i:Y_i=y} x_{i,j} \\
\phi_{y,j} \propto& \sum_{i:Y_i=y} x_{i,j} = \sum_i \delta(Y_i = y) x_{i,j} \\
=& \frac{\sum_{i:Y_i=y} x_{i,j}}{\sum_{j'} \sum_{i:Y_i=y} x_{i,j'}}
\end{aligned}$$

Similarly, $\mu_y \propto \sum_i \delta(Y_i = y)$, where $\delta(Y_i = y) = 1$ if $Y_i = y$ and 0 otherwise.

Smoothing and MAP estimation

If data is sparse, you can end up with values of $\phi = 0$, allowing a single feature to completely veto a label. 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 Laplace smoothing: adding "pseudo-counts" of α to each estimate, and then normalize.

$$\phi_{y,j} = \frac{\alpha + \sum_{i:Y_i=y} x_{i,j}}{\sum_{j'} \alpha + \sum_{i:Y_i=y} x_{i,j'}} = \frac{\alpha + \text{count}(i,j)}{V\alpha + \sum_{j'} \text{count}(i,j')} \tag{3.20}$$

Laplace smoothing has a nice Bayesian justification, in which we extend the generative story to include ϕ 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** (towards uniform probabilities). Machine learning theory shows that errors on held out data result from the sum of bias and variance. Techniques for reducing variance typically increase the bias, so there is a **bias-variance tradeoff**.

- 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 just go for low bias.

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 do this, 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 we're likely to get on new data. See `http://xkcd.com/1122/` for a cartoon related to this idea.

Many learning algorithms also have "tuning" parameters:

- the smoothing pseudo-counts α in Naive Bayes
- the regularization λ in logistic regression
- the slack weight C in the support-vector machine

All of these tuning parameters really do the same thing: they navigate the bias-variance tradeoff. 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. Instead, you should split off a piece of your training data, called a "development set" (or "tuning set").

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

The Naivety of Naive Bayes

Naive Bayes is very simple to work with. Estimation and prediction can done in closed form, and the nice probabilistic interpretation makes it relatively easy to extend the model in various ways.

But Naive 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:

$$\mathrm{p}(\textit{naïve}, \textit{Bayes}; \phi) = \mathrm{p}(\textit{naïve}; \phi)\mathrm{p}(\textit{Bayes}; \phi). \tag{3.21}$$

- But this is clearly wrong, because words "travel together." Question for you, is it:

$$\mathrm{p}(\textit{naïve Bayes}) > \mathrm{p}(\textit{naïve})\mathrm{p}(\textit{Bayes}) \tag{3.22}$$

  or...

$$\mathrm{p}(\textit{naïve Bayes}) < \mathrm{p}(\textit{naïve})\mathrm{p}(\textit{Bayes}) \tag{3.23}$$

  Apply the chain rule!

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

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

Now, suppose you see that both lights are red. According to Naive 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 $P(R, R \mid \text{not-broken}) = P_{\text{north-south}}(R \mid \text{not-broken}) P_{\text{east-west}}(R \mid \text{not-broken})$. But this independence assumption is clearly incorrect, because $P(R, R \mid \text{not-broken}) = 0$.

Less Naive Bayes? Of course we could decide not to make the naive Bayes assumption, and model $P(R, R)$ 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'd be fine (in theory, maybe not in practice); but we never have that. Naive Bayes accepts some bias (because of the incorrect modeling assumption) in exchange for lower variance.

3.3 Recap

- Bag-of-words representation $\boldsymbol{f}(\boldsymbol{x}, y)$
- Classification as a dot-product $\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}, y)$
- Naive Bayes
  - Define $\mathrm{p}(\boldsymbol{x}, \boldsymbol{y})$ via a *generative model*
  - Prediction: $\hat{y} = \arg\max_y \mathrm{p}(\boldsymbol{x}_i, y)$
  - Learning:

$$
\begin{aligned}
\boldsymbol{\theta} =& \arg\max_{\boldsymbol{\theta}} \mathrm{p}(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{\theta}) \\
\mathrm{p}(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{\theta}) =& \prod_i \mathrm{p}(\boldsymbol{x}_i, y_i; \boldsymbol{\theta}) = \prod_i \mathrm{p}(\boldsymbol{x}_i | y_i) \mathrm{p}(y_i) \\
\phi_{y,j} =& \frac{\sum_{i:Y_i=y} x_{ij}}{\sum_{i:Y_i=y} \sum_j x_{ij}} \\
\mu_y =& \frac{\text{count}(Y = y)}{N}
\end{aligned}
$$

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

- Bias-variance tradeoff: MLE is high-variance, so add smoothing pseudo counts α. This reduces variance but adds bias.

Chapter 4

Sentiment analysis

Todo: add notes about sentiment analysis here

Chapter 5

Discriminative learning

5.1 Features

Naive Bayes is a simple classifier, where the weights are learned based on the joint probability of labels and words. It includes an independence assumption: all features are mutually independent, conditioned on the label.

- We have defined a **feature function** $\boldsymbol{f}(\boldsymbol{x}, y)$, which corresponds to "bag-of-words" features. While these features do violate the independence assumption, the violation is relatively mild.
- We may be interested in other features, which violate independence more severely. Can you think of any?
  - Prefixes, e.g. *anti-*, *im-*, *un-*
  - Punctuation and capitalization
  - Bigrams, e.g. *not good*, *not bad*, *least terrible*, ...

Rich feature sets generally cannot be combined with Naive Bayes because the distortions resulting from violations of the independence assumption overwhelm the additional power of better features.

$$\mathrm{p}(\textit{not bad food}|y) \approx \mathrm{p}(\textit{not}|y)\mathrm{p}(\textit{bad}|y)\mathrm{p}(\textit{food}|y) \tag{5.1}$$
$$\mathrm{p}(\textit{not bad food}|y) \not\approx \mathrm{p}(\textit{not}|y)\mathrm{p}(\textit{bad}|y)\mathrm{p}(\textit{not bad}|y)\mathrm{p}(\textit{food}|y) \tag{5.2}$$

To use these features, we will need learning algorithms that do not rely on an independence assumption.

5.2 Perceptron

In NB, 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?

- Until converged, at each iteration t
  - Select an instance i
  - Let $\hat{y} = \arg\max_y \boldsymbol{\theta}_t^\top \boldsymbol{f}(\boldsymbol{x}_i, y)$
  - If $\hat{y} = y_i$, do nothing
  - If $\hat{y} \neq y_i$, set $\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t + \boldsymbol{f}(\boldsymbol{x}_i, y_i) - \boldsymbol{f}(\boldsymbol{x}_i, \hat{y})$

Basically we are saying: 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 seems like a cheap heuristic, right? Will it really work? In fact, there is some nice theory for the perceptron.

- If there is a set of weights that correctly separates your data, then your data is **separable**.
- Formally, your data is (linearly) separable if there exists a set of weights $\boldsymbol{\theta}$ such that
$$\forall \boldsymbol{x}_i, y_i,\ \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}_i, y_i) > \max_{y' \neq y_i} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}_i, y') \tag{5.3}$$
- If your data is linearly separable, it can be proven that the perceptron algorithm will eventually find a separator.
- What if your data is not separable?
  - the number of errors is bounded...
  - but the algorithm will thrash. That is, the weights will cycle between different values, and will never converge.

The perceptron is an **online** learning algorithm.

- This means that it adjusts the weights after every example.

- This is different from Naïve Bayes, which computes corpus statistics and then sets the weights in a single operation. This is a **batch learning** algorithm.
- Other algorithms are **iterative**, in that they perform multiple updates to the weights, but are also **batch**, in that they have to use all the training data to compute the update. We'll mention two of those algorithms later.

Voted (averaged) perceptron

One solution to thrashing is to average the weights across all iterations:

$$\begin{aligned}\overline{\boldsymbol{\theta}} =& \frac{1}{T}\sum_{t=1}^{T}\boldsymbol{\theta}_t \\ y =& \arg\max_y \overline{\boldsymbol{\theta}}^\top \boldsymbol{f}(\boldsymbol{x}, y)\end{aligned}$$

There is some analysis showing that voting can improve generalization (Freund and Schapire, 1999; Collins, 2002). However, this rule as described here is not practical. Can you see why not, and how to fix it?

5.3 Loss functions and large-margin classification

Naive Bayes chooses the weights $\boldsymbol{\theta}$ by maximizing the likelihood $\mathrm{p}(\boldsymbol{x}, \boldsymbol{y})$. This can be seen, equivalently, as maximizing the log-likelihood (due to the monotonicity of the log function), and as **minimizing** the negative log-likelihood. This negative log-likelihood can therefore be viewed as a **loss function**, which is minimized:

$$\log \mathrm{p}(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{\theta}) = \sum_{i=1}^{N} \log \mathrm{p}(\boldsymbol{x}_i, y_i; \boldsymbol{\theta}) \tag{5.4}$$

$$\ell_{\mathrm{NB}}(\boldsymbol{\theta}; \boldsymbol{x}_i, y_i) = -\log \mathrm{p}(\boldsymbol{x}_i, y_i; \boldsymbol{\theta}) \tag{5.5}$$

$$\hat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \sum_{i=1}^{N} \ell_{\mathrm{NB}}(\boldsymbol{\theta}, \boldsymbol{x}_i, y_i) \tag{5.6}$$

This may seem confusing and backwards, but loss functions provide a very general framework in which to compare many approaches to machine learning.

For example, even though the perceptron is not a probabilistic model, it is also trying to minimize a **loss function**:

$$\ell_{\text{perceptron}}(\boldsymbol{\theta}; \boldsymbol{x}_i, y_i) = \begin{cases} 0, & y_i = \arg\max_y \boldsymbol{\theta}^\top \boldsymbol{f}(x_i, y) \\ 1, & \text{otherwise} \end{cases} \tag{5.7}$$

This loss function has some pros and cons in comparison with Naive Bayes.

- ℓ_{NB} can suffer **infinite** loss on a single example, which suggests it will overemphasize some examples, and underemphasize others.
- $\ell_{\text{perceptron}}$ treats all errors equally. It only cares if the example is correct, and not about how confident the classifier was. Since we usually evaluate on accuracy, this is a better match.
- $\ell_{\text{perceptron}}$ is non-convex[1] and discontinuous. Finding the global optimum is intractable when the data is not separable.

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

$$\gamma(\boldsymbol{\theta}; \boldsymbol{x}_i, y_i) = \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}_i, y_i) - \max_{y \neq y_i} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}_i, y) \tag{5.8}$$

Then we can write a convex and continuous "hinge loss" as

$$\ell_{\text{hinge}}(\boldsymbol{\theta}; \boldsymbol{x}_i, y_i) = \begin{cases} 0, & \gamma(\boldsymbol{\theta}; \boldsymbol{x}_i, y_i) \geq 1, \\ 1 - \gamma(\boldsymbol{\theta}; \boldsymbol{x}_i, y_i), & \text{otherwise} \end{cases} \tag{5.9}$$

Equivalently, we can write $\ell_{\text{hinge}}(\boldsymbol{\theta}; \boldsymbol{x}_i, y_i) = (1 - \gamma(\boldsymbol{\theta}; \boldsymbol{x}_i, y_i))_+$, where $(x)_+$ indicates the positive part of x.

Essentially, we want a *margin* of at least 1 between the score for the true label and the best-scoring alternative, which we have written $\hat{y}$.

The hinge and perceptron loss functions are shown in Figure 5.1.

[1] As a reminder, 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 a wide array of techniques for optimizing convex functions (Boyd and Vandenberghe, 2004)

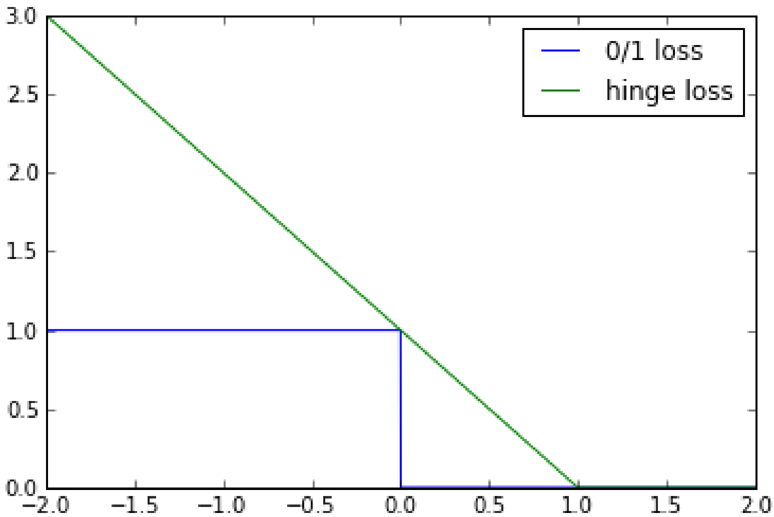

Figure 5.1: Hinge and perceptron loss functions

Large-margin online classification

Note that we can write $\boldsymbol{\theta} = s\boldsymbol{u}$, where $||\boldsymbol{u}||_2 = 1$. 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 which attain zero hinge loss. For generality, we will try to make the smallest magnitude change to $\boldsymbol{\theta}$ possible.[2]

At step t, we optimize:

$$\boldsymbol{\theta}_{t+1} = \arg\min_{\boldsymbol{\theta}} \frac{1}{2}||\boldsymbol{\theta} - \boldsymbol{\theta}_t||^2 \; s.t. \; \ell_{\text{hinge}}(\boldsymbol{\theta}; \boldsymbol{x}_i, y_i) = 0 \tag{5.10}$$

Assuming that the constraint can be satisfied (i.e., the problem is linearly separabe), the optimal solution is found at,

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \tau_t(\boldsymbol{f}(y_i, \boldsymbol{x}_i) - \boldsymbol{f}(\hat{y}, \boldsymbol{x}_i)) \tag{5.11}$$

$$\tau_t = \frac{\ell(\boldsymbol{\theta}; \boldsymbol{x}_i, y_i)}{||\boldsymbol{f}(x_i, y_i) - \boldsymbol{f}(x_i, \hat{y})||^2}, \tag{5.12}$$

[2]In the support vector machine (without slack variables), we choose the smallest magnitude weights that satisfy the constraint of zero hinge loss. Pegasos is an online algorithm for training SVMs (Shwartz et al., 2007); it is similar to Passive-Aggressive.

where again $\hat{y}$ is the best scoring y according to $\boldsymbol{\theta}_t$. This solution can be obtained by introducing τ_t as a Lagrange multiplier for the constraint in (5.10).

If the data is not lienarly separable, there will be instances for which we can't meet this constraint. To deal with this, we introduce a "slack" variable ξ_i. We use the slack variable to trade off between the constraint (having a large margin) and the objective (having a small change in $\boldsymbol{\theta}$). The tradeoff is controlled by a parameter C.

$$\begin{aligned} &\min w \frac{1}{2}||\boldsymbol{\theta} - \boldsymbol{\theta}_t||^2 + C\xi_t \\ &s.t. \ell_{\text{hinge}}(\boldsymbol{\theta}; \boldsymbol{x}_i, y_i) \leq \xi_t, \xi_t \geq 0 \end{aligned} \tag{5.13}$$

The solution to 5.13 is,

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \tau_t(\boldsymbol{f}(y_i, \boldsymbol{x}_i) - \boldsymbol{f}(\hat{y}, \boldsymbol{x}_i)) \tag{5.14}$$

$$\tau_t = \min\left(C, \frac{\ell(\boldsymbol{\theta}; \boldsymbol{x}_i, y_i)}{||\boldsymbol{f}(\boldsymbol{x}_i, y_i) - \boldsymbol{f}(\boldsymbol{x}_i, \hat{y})||^2}\right), \tag{5.15}$$

- If C is 0, then infinite slack is permitted, and the weights will never change.

- As $C \to \infty$, no slack is permitted, and the optimization is identical to equation 5.10 and 5.12.

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.[3]

- PA is error-driven like the perceptron, but is more stable to violations of separability, like the averaged perceptron.

- 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.

- You can also apply weight averaging to PA.

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

- **Support vector machines** (SVMs) are another learning algorithm based on the hinge loss (Burges, 1998), but they try to minimize the norm of the weights, rather than the norm of the change in the weights. They are typically trained in **batch** style, meaning that they have to read all the training instances in to compute each update. However, SVMs can also be trained in an online fashion (Shwartz et al., 2007). The LXMLS lab guide provides a simpler online learning algorithm, based on stochastic subgradient descent (Figueiredo et al., 2013).

Pros and cons of Perceptron and PA

- Perceptron and PA are error-driven, which means they usually do better in practice than naive Bayes.
- They are also online, which means we can learn without having our whole dataset in memory at once. NB can also be estimated online, in the sense that you can stream the data and store the counts.
- The original perceptron doesn't behave well if the data is not separable, and doesn't make it easy to control model complexity.
- All these models lack a probabilistic interpretation. Probabilities are useful because they quantify the classification certainty, allowing us to compute expected utility, and to incorporate the classifier in more complex probabilistic models.

5.4 Logistic regression

Logistic regression is error-driven like the perceptron, but probabilistic like Naive Bayes. This is useful in case we want to quantify the uncertainty about a classification decision.

Recall that NB selects weights to optimize the joint probability $\mathrm{p}(y, \boldsymbol{x})$.

- In NB, we factor this as $\mathrm{p}(y, \boldsymbol{x}) = \mathrm{p}(\boldsymbol{x}|y)\mathrm{p}(y)$.
- But we could equivalently write $\mathrm{p}(y, \boldsymbol{x}) = \mathrm{p}(y|\boldsymbol{x})\mathrm{p}(\boldsymbol{x})$.

Since we always know $\boldsymbol{x}$, we really care only about $\mathrm{p}(y|\boldsymbol{x})$. Logistic regression optimizes this directly. To do this, we have to define the probability function differently. We define the conditional probability directly, as,

$$\mathrm{p}(y|\boldsymbol{x}) = \frac{\exp\left(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}, y)\right)}{\sum_{y' \in \mathcal{Y}} \exp\left(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}, y')\right)} \tag{5.16}$$

$$\log \mathrm{p}(y|\boldsymbol{x}) = \sum_{i=1}^{N} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}_i, y_i) - \log \sum_{y' \in \mathcal{Y}} \exp \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}_i, y') \tag{5.17}$$

$$\hat{\boldsymbol{\theta}} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{N} \log \mathrm{p}(y_i|\boldsymbol{x}_i; \boldsymbol{\theta}) \tag{5.18}$$

Inside the sum, we have the (additive inverse of) the **logistic loss**.

- In binary classification, we can write this as

$$\ell_{\text{logistic}}(\boldsymbol{\theta}; \boldsymbol{x}_i, y_i) = -(y_i \boldsymbol{\theta}^\top \boldsymbol{x}_i - \log\left(1 + \exp \boldsymbol{\theta}^\top \boldsymbol{x}_i\right)) \tag{5.19}$$

- In multi-class classification, we have,[4]

$$\ell_{\text{logistic}}(\boldsymbol{\theta}; \boldsymbol{x}_i, y_i) = -(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}_i, y_i) - \log \sum_{y' \in \mathcal{Y}} \exp \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}_i, y')) \tag{5.20}$$

The logistic loss is shown in Figure 5.2. Because it is smooth and convex, we can optimize it through gradient steps:

[4]The log-sum-exp term is very common in machine learning. It is numerically instable because you can underflow if the inner product is small, and overflow if the inner product is large. Libraries like `scipy` 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.

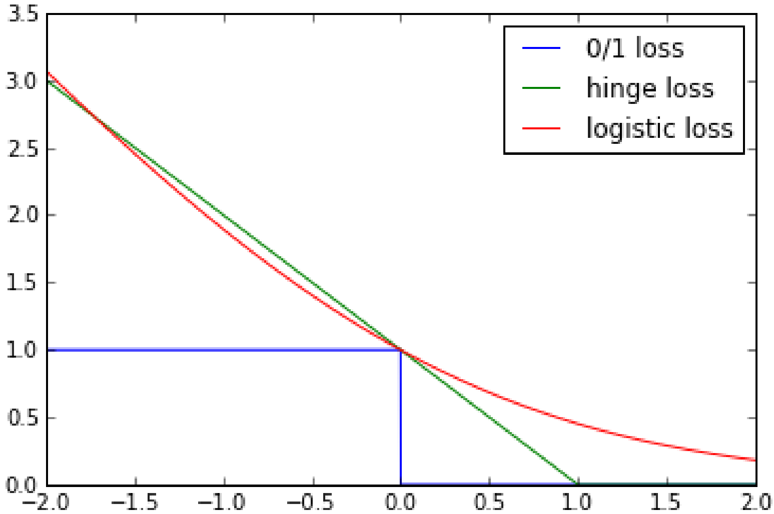

Figure 5.2: Hinge, perceptron, and logistic loss functions

$$\ell = \sum_{i=1}^{N} \boldsymbol{\theta}^{\mathsf{T}} \boldsymbol{f}(\boldsymbol{x}_i, y_i) - \log \sum_{y' \in \mathcal{Y}} \exp \boldsymbol{\theta}^{\mathsf{T}} \boldsymbol{f}(\boldsymbol{x}_i, y') \tag{5.21}$$

$$\frac{\partial \ell}{\partial \boldsymbol{\theta}} = \sum_{i=1}^{N} \boldsymbol{f}(\boldsymbol{x}_i, y_i) - \frac{\sum_{y' \in \mathcal{Y}} \exp \boldsymbol{\theta}^{\mathsf{T}} \boldsymbol{f}(\boldsymbol{x}_i, y') \boldsymbol{f}(\boldsymbol{x}_i, y')}{\sum_{y'' \in \mathcal{Y}} \exp \boldsymbol{\theta}^{\mathsf{T}} \boldsymbol{f}(\boldsymbol{x}_i, y'')} \tag{5.22}$$

$$= \sum_{i=1}^{N} \boldsymbol{f}(\boldsymbol{x}_i, y_i) - \sum_{y' \in \mathcal{Y}} \frac{\exp \boldsymbol{\theta}^{\mathsf{T}} \boldsymbol{f}(\boldsymbol{x}_i, y')}{\sum_{y'' \in \mathcal{Y}} \exp \boldsymbol{\theta}^{\mathsf{T}} \boldsymbol{f}(\boldsymbol{x}_i, y'')} \boldsymbol{f}(\boldsymbol{x}_i, y') \tag{5.23}$$

$$= \sum_{i=1}^{N} \boldsymbol{f}(\boldsymbol{x}_i, y_i) - \sum_{y' \in \mathcal{Y}} \mathrm{p}(y' | \boldsymbol{x}_i; \boldsymbol{\theta}) \boldsymbol{f}(\boldsymbol{x}_i, y') \tag{5.24}$$

$$= \sum_{i=1}^{N} \boldsymbol{f}(\boldsymbol{x}_i, y_i) - E[\boldsymbol{f}(\boldsymbol{x}_i, y)] \tag{5.25}$$

This gradient has a pleasing interpretation as the difference between the observed counts and the expected counts.[5] Compare this gradient with the percep-

[5]Recall that the definition of an expected value $E[f(x)] = \sum_x f(x)\mathrm{p}(x)$

tron and PA update rules.

The bias-variance tradeoff is handled by penalizing large $\boldsymbol{\theta}$ in the objective, adding a term of $\frac{\lambda}{2}||\boldsymbol{\theta}||_2^2$. This is called L2 regularization, because of the L2 norm. It can be viewed as placing a 0-mean Gaussian prior on $\boldsymbol{\theta}$.

This penalty contributes a term of $\lambda\boldsymbol{\theta}$ to the gradient, so we have,

$$\ell = \sum_{i=1}^{N} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}_i, y_i) - \log \sum_{y' \in \mathcal{Y}} \exp \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}_i, y') + \frac{\lambda}{2}||\boldsymbol{\theta}||_2^2 \tag{5.26}$$

$$\frac{\partial \ell}{\partial \boldsymbol{\theta}} = \sum_{i=1}^{N} \boldsymbol{f}(\boldsymbol{x}_i, y_i) - E[\boldsymbol{f}(\boldsymbol{x}_i, y)] - \lambda\boldsymbol{\theta}. \tag{5.27}$$

Optimization

Batch optimization In batch optimization, you keep all the data in memory and iterate over it many times.

- The logistic loss is smooth and convex, so we can find the global optimum using gradient descent. But in practice, this can be very slow.

- Second-order (Newton) optimization would incorporate the inverse Hessian. The Hessian is
$$H_{i,j} = \frac{\partial^2}{\partial w_i \partial w_j}\ell, \tag{5.28}$$
but this matrix is usually too big to deal with.

- In practice, people usually apply **quasi-Newton optimization**, which approximates the Hessian matrix. The specific method that is particularly popular is L-BFGS[6] NLP people usually treat L-BFGS as a black box; you will typically pass it a pointer to a function that computes the likelihood and gradient. L-BFGS is provided in `scipy.optimize`.

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

[6] A friend of mine told me you can remember the order of the letters as "Large Big Friendly Giants." Does this help you?

stochastic online approximation to the overall gradient:

$$\theta^{(t+1)} \leftarrow \theta^{(t)} - \eta_t \nabla_{\theta} \ell(\theta^{(t)}, x, y) \tag{5.29}$$

$$= \theta^{(t)} - \eta_t (\lambda \theta^{(t)} - \sum_i^N f(x_i, y_i) - E[f(x_i, y)]) \tag{5.30}$$

$$= (1 - \lambda\eta_t)\theta^{(t)} + \eta_t \sum_i^N f(x_i, y_i) - E[f(x_i, y)] \tag{5.31}$$

$$\approx (1 - \lambda\eta_t)\theta^{(t)} + N\eta_t \left(f(x_{i(t)}, y_{i(t)}) - E[f(x_{i(t)}, y)] \right) \tag{5.32}$$

where η_t is the **stepsize** at iteration t, and $\langle x_{i(t)}, y_{i(t)} \rangle$ is the instance selected at iteration t. (So here we are setting the mini-batch size equal to one.) As always, N is the total number of instances. As above, the expectation is equal to a weighted sum over the labels,

$$E[f(x_{i(t)}, y)] = \sum_{y' \in \mathcal{Y}} \mathrm{p}(y' | x_{i(t)}; \theta) f(x_{i(t)}, y'). \tag{5.33}$$

- Note how similar this update is to the perceptron!
- If we set $\eta_t = \eta_0 t^{-\alpha}$ for $\alpha \in [1, 2]$, we have guaranteed convergence.
- We can also just fix η_t to a small value, like 10^-3. (This is what we will do in the problem set.)
- In either case, we could tune this parameter on a development set. However, it would be acceptable to just find a value that gives a good regularized log-likelihood on the training set, since this parameter relates to the quality of the optimization, and not the generalization capability of the classifier.
- In theory, we select $\langle x_{i(t)}, y_{i(t)} \rangle$ at random, but in practice we usually just iterate through the dataset.
- We can fold N into η and λ, so that $\eta^* = N\eta$ and $\lambda^* = \lambda \frac{\eta^*}{N}$. This gives the more compact form,

$$(1 - \lambda^* \eta_t^*)\theta^{(t)} + \eta_t^* \left(f(x_{i(t)}, y_{i(t)}) - E[f(x_{i(t)}, y)] \right) \tag{5.34}$$

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 a 1978 paper by GT's own Arkadi Nemirovski (Nemirovski and Yudin, 1978). You can find several recent chapters about online optimization in the edited volume by Sra et al. (2012).

Adagrad Recent work has shown that you can often learn more quickly by using an **adaptive** step-size, which is different for every feature (Duchi et al., 2011). Specifically, in the **Adagrad** algorithm (adaptive gradient), you keep track of the sum of the squares of the gradients for each feature, and rescale the learning rate by its inverse:

$$\boldsymbol{g}_t = - \boldsymbol{f}(\boldsymbol{x}_i, y_i) + \sum_{y' \in \mathcal{Y}} \mathrm{p}(y' \mid \boldsymbol{x}_i) \boldsymbol{f}(\boldsymbol{x}_i, y_i) + \lambda \boldsymbol{\theta} \tag{5.35}$$

$$\theta_j^{(t+1)} \leftarrow \theta_j^{(t)} - \frac{\eta}{\sqrt{\sum_{t'=1}^{t} g_{t,j}^2}} g_{t,j}, \tag{5.36}$$

where j iterates over features in $\boldsymbol{f}(\boldsymbol{x}, y)$. The effect of this is that features with consistently large gradients are updated more slowly. Another way to view this update is that rare features are taken more seriously, since their sum of squared gradients will be smaller. Adagrad seems to require less careful tuning of η, and Dyer (2014) reports that $\eta = 1$ works for a wide range of problems.

Note that the Adagrad update can apply to any smooth loss function, including the hinge loss defined in Equation 5.9.

Names

Logistic regression is so named because in the binary case where $y \in \{0, 1\}$, we are performing a regression of $\boldsymbol{x}$ against $\boldsymbol{y}$, after passing the inner product $\boldsymbol{\theta}^\top \boldsymbol{x}$ through a logistic transformation. You could always do a linear regression, but this would ignore the fact that the $\boldsymbol{y}$ is limited to a few values.

- Logistic regression is also called **maximum conditional likelihood** (MCL), because it maximizes... the conditional likelihood $\mathrm{p}(y \mid \boldsymbol{x})$.

- Logistic regression can be viewed as part of a larger family, called **generalized linear models**. If you use R, you are probably familiar with `glmnet`.

- Logistic regression is also called **maximum entropy**, especially in the earlier NLP literature (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:

$$\forall j, \sum_{i=1}^{N} f_j(\boldsymbol{x}_i, y_i) = \sum_{i=1}^{N} \sum_{y \in \mathcal{Y}} \mathrm{p}(y \mid \boldsymbol{x}_i; \boldsymbol{\theta}) f_j(\boldsymbol{x}_i, y) \tag{5.37}$$

Note that this constraint will be met exactly when the derivative of the likelihood function (equation 5.25) is equal to zero. However, this will be true for many values of $\boldsymbol{\theta}$. Which should we choose?

The entropy of a conditional likelihood function $P(Y|X)$ is

$$H(P) = -\sum_{x \in \mathcal{X}} \tilde{\mathrm{p}}(x) \sum_{y \in \mathcal{Y}} \mathrm{p}(y|x) \log \mathrm{p}(y|x), \tag{5.38}$$

where $\tilde{\mathrm{p}}(x)$ is the *empirical probability* of x. We compute an empirical probability by summing over all the instances in training set.

If the entropy is large, this function is smooth across possible values of y; if it is small, the function is sharp. The entropy is zero if $\mathrm{p}(y|x) = 1$ for some particular $Y = y$ and zero for everything else. By saying we want maximum-entropy classifier, we are saying we want to make the least commitments possible, while satisfying the moment-matching constraints:

$$\max_{\boldsymbol{\theta}} \quad -\sum_{\boldsymbol{x}} \tilde{\mathrm{p}}(\boldsymbol{x}) \sum_{y} \mathrm{p}(y|\boldsymbol{x}; \boldsymbol{\theta}) \log \mathrm{p}(y|\boldsymbol{x}; \boldsymbol{\theta}) \tag{5.39}$$

$$s.t. \quad \forall j, \sum_{i=1}^{N} f_j(\boldsymbol{x}_i, y_i) = \sum_{i=1}^{N} \sum_{y} \mathrm{p}(y|\boldsymbol{x}_i; \boldsymbol{\theta}) f_j(\boldsymbol{x}_i, y) \tag{5.40}$$

Now, the solution to this constrained optimization problem is identical to the maximum conditional likelihood (logistic-loss) formulation we've considered in the previous section.

This view of logistic regression is arguably a little dated, but it's useful to understand what's going on. The information-theoretic concept of entropy will pop up again a few times in the course. For a tutorial on maximum entropy, see `http://www.cs.cmu.edu/afs/cs/user/aberger/www/html/tutorial/tutorial.html`.

5.5 Summary of learning algorithms

- **Naive Bayes**. pros: easy and probabilistic. cons: arguably optimizes wrong objective; usually has poor accuracy, especially with overlapping features.

- **Perceptron and PA**. pros: easy, online, and error-driven. cons: not probabilistic. this can be bad in pipeline architectures, where the output of one system becomes the input for another.

- **Logistic regression**. pros: error-driven and probabilistic. cons: batch learning requires black-box software; hinge loss sometimes yields better accuracy than logistic loss.

What about non-linear classification?

The feature spaces that we consider in NLP are usually huge, so non-linear classification can be quite difficult. When the feature dimension V is larger than the number of instances N — often the case in NLP — you can always learn a linear classifier that will perfectly classify your training instances.[7] This makes selecting an appropriate **non-linear** classifier especially difficult. Nonetheless, there are some approaches to non-linear learning in NLP:

- You can add **features**, such as bigrams, which are non-linear combinations of other features. For example, the base feature $\langle$*coffee house*$\rangle$ will not fire unless both features $\langle$*coffee*$\rangle$ and $\langle$*house*$\rangle$ also fire.

- Another option is to apply non-linear transformations to the feature vector. Recall that the feature function $\boldsymbol{f}(\boldsymbol{x}, y)$ may be composed of a vector of word counts, padded by zeros. We can think of these word counts as basic features, and apply non-linear transformations, such as $\boldsymbol{x} \circ \boldsymbol{x}$ or $|\boldsymbol{x}|$.

- There is some work in NLP on using kernels for strings, bags-of-words, sequences, trees, etc. Kernelized learning algorithms are outside the scope of this class (Collins and Duffy, 2001; Zelenko et al., 2003). Kernel-based learning can be seen as a generalization of algorithms such k-nearest-neighbors, which classifies instances by considering the labels of the k most similar instances in the training set (Hastie et al., 2009).

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

- Boosting (Freund et al., 1999) and decision tree algorithms (Schmid, 1994) sometimes do well on NLP tasks, but they are used less frequently these days, especially as the field increasingly emphasizes big data and simple classifiers.

- More recent work has shown how **deep learning** can perform non-linear classification. One way to use deep learning in NLP is by learning word representations while jointly learning how these representations combine to classify instances (Collobert and Weston, 2008). This approach is very hot at the moment, so I will discuss it towards the end of the semester.

5.6 Summary of classifiers

So now we've talked about four different classifiers. That's it! No more classifiers in this class. Yay? Anyway, let's review.

	Naive Bayes	Logistic Regression	Perceptron	PA
Objective	Joint likelihood	Conditional likelihood	0-1 loss	Hinge loss
	$\max \sum_i \log \mathrm{p}(\boldsymbol{x}_i, y_i)$	$\max \sum_i \log \mathrm{p}(y_i \vert \boldsymbol{x}_i)$	$\min \sum_i \delta(y_i, \hat{y})$	$\sum_i [1 - \gamma(\boldsymbol{\theta}; \boldsymbol{x}_i, y_i)]_+$
estimation	$\theta_{ij} = \frac{c(x_i, y=j)+\alpha}{c(y=j)+V\alpha}$	$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}} = \sum_i \boldsymbol{f}(\boldsymbol{x}_i, y_i) - E[\boldsymbol{f}(\boldsymbol{x}_i, y)]$	$\boldsymbol{\theta}^{(t)} \leftarrow \boldsymbol{\theta}^{(t-1)} + \boldsymbol{f}(\boldsymbol{x}_i, y_i) - \boldsymbol{f}(\boldsymbol{x}_i, \hat{y})$	$\boldsymbol{\theta}^{(t)} \leftarrow \boldsymbol{\theta}^{(t-1)} + \tau_t(\boldsymbol{f}(\boldsymbol{x}_i, y_i) - \boldsymbol{f}(\boldsymbol{x}_i, \hat{y}))$
tuning	smoothing α	regularizer $\lambda \vert\vert \boldsymbol{\theta} \vert\vert_2^2$	weight averaging	slack penalty C
complexity	$\mathcal{O}(NV)$	$\mathcal{O}(NVT)$	$\mathcal{O}(NVT)$	$\mathcal{O}(NVT)$
easy?	very	not really	yes	yes
probabilities?	yes	yes	no	no
features?	no	yes	yes	yes

Table 5.1: Comparison of classifiers. N = number of examples, V = number of features, T = number of instances.

Chapter 6

Word-sense disambiguation

Todo: add notes about WSD here

Chapter 7

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, you get a bunch of text, and you suspect that there are at least two different meanings for the word *concern*.[1]

The immediate context includes two groups of words:

- services, produces, banking, pharmaceutical, energy, electronics
- about, said, that, over, in, with, had

Suppose we plot each instance of *concern* on a graph

- x-axis is the density of words in group 1
- y-axis is the density of words in group 2

Two blobs might emerge. These blobs would correspond to two different sense of *concern*.

- But in reality, we don't know the word groupings in advance.

[1] example from Pedersen and Bruce (1997)

- 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!)
- Or we have to automatically find a low-dimensional projection. More on that much later in the course.

Here's a related scenario:

- You look at thousands of news articles from today
- Plot them on a graph of *Miley* vs *Syria*
- Three clumps emerge (Miley, Syria, others)
- Those clumps correspond to natural document classes
- Again, in reality this is a hugely high-dimensional graph

So these examples show that we can find structure in data, even without labels.

7.1 K-means clustering

You might know about classic clustering algorithms like K-means. These algorithms are iterative:

1. Guess the location of cluster centers.
2. Assign each point to the nearest center.
3. Re-estimate the centers as the mean of the assigned points.
4. Goto 2.

This is an algorithm for finding coherent "blobs" of documents.
There is a variant called "soft k-means."

- Instead of assigning each point $\boldsymbol{x}_i$ to a specific cluster z_i
- You assign it a distribution over clusters $q_i(z_i)$

We're now going to explore a more principled, statistical version of soft K-means, called EM clustering.

By understanding the statistical principles underlying the algorithm, we can extend it in a number of cool ways.

7.2 The Expectation-Maximization Algorithm

Let's go back to the Naive Bayes model:

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

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 something?

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

$$\begin{aligned} \log \mathrm{p}(\boldsymbol{x}) &= \log \sum_{\boldsymbol{y}} \mathrm{p}(\boldsymbol{x} | \boldsymbol{y}; \boldsymbol{\phi}) \mathrm{p}(\boldsymbol{y}; \mu) && (7.1) \\ &= \log \sum_{\boldsymbol{y}} \prod_i \mathrm{p}(\boldsymbol{x}_i | y_i; \boldsymbol{\phi}) \mathrm{p}(y_i; \mu) && (7.2) \\ &= \sum_i \log \sum_{y_i} \mathrm{p}(\boldsymbol{x}_i | y_i; \boldsymbol{\phi}) \mathrm{p}(y_i; \mu) && (7.3) \end{aligned}$$

Now we introduce an auxiliary variable $\boldsymbol{q}_i$, for each y_i. 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 Y, for each instance i.

Now since $\frac{q_i(y)}{q_i(y)} = 1$,

$$\begin{aligned} \log \mathrm{p}(\boldsymbol{x}) &= \sum_i \log \sum_{y_i} \mathrm{p}(\boldsymbol{x}_i | y_i; \boldsymbol{\phi}) \mathrm{p}(y_i; \mu) \frac{q_i(y)}{q_i(y)} \\ &= \sum_i \log E_q [\frac{\mathrm{p}(\boldsymbol{x}_i | y; \boldsymbol{\phi}) \mathrm{p}(y; \mu)}{q_i(y)}], \end{aligned}$$

by the definition of expectation. (Note that E_q just means the expectation under the distribution q.)

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

$$\begin{aligned} \log \mathrm{p}(\boldsymbol{x}) &\geq \sum_i E_q [\log \frac{\mathrm{p}(\boldsymbol{x}_i | y; \boldsymbol{\phi}) \mathrm{p}(y_i; \mu)}{q_i(y)}] \\ \mathcal{J} &= \sum_i E_q [\log \mathrm{p}(\boldsymbol{x}_i | y; \boldsymbol{\phi})] + E_q [\log \mathrm{p}(y; \mu)] - E_q [q_i(y)] \end{aligned}$$

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 arguments:

- the distributions $q_i(\boldsymbol{y})$ 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.

The E-step

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

$$
\begin{aligned}
\mathcal{J} =& \sum_i E_q[\log \mathrm{p}(\boldsymbol{x}_i | y; \boldsymbol{\phi})] + E_q[\log \mathrm{p}(y; \mu)] - E_q[q_i(y)] \\
=& \sum_i \sum_y q_i(y) \left(\log \mathrm{p}(\boldsymbol{x}_i | Y_i = y; \boldsymbol{\phi}) + \log \mathrm{p}(y; \mu) - \log q_i(y)\right)
\end{aligned}
$$

As in relative frequency estimation of Naive Bayes, we need to add a Lagrange multiplier to ensure $\sum_y q_i(y) = 1$, so

$$
\begin{aligned}
\mathcal{J} =& \sum_i \sum_y q_i(y) \left(\log \mathrm{p}(\boldsymbol{x}_i | Y_i = y; \boldsymbol{\phi}) + \log \mathrm{p}(y; \mu) - \log q_i(y)\right) + \lambda_i (1 - \sum_y q_i(y)) \\
\frac{\partial \mathcal{J}}{\partial q_i(y)} =& \log \mathrm{p}(\boldsymbol{x}_i | Y_i = y; \boldsymbol{\phi}) + \log \mathrm{p}(y; \boldsymbol{\theta}) - \log q_i(y) - 1 - \lambda_i \\
\log q_i(y) =& \log \mathrm{p}(\boldsymbol{x}_i | Y_i = y; \boldsymbol{\phi}) + \log \mathrm{p}(y; \mu) - 1 - \lambda_i \\
q_i(y) \propto& \mathrm{p}(\boldsymbol{x}_i | Y_i = y; \boldsymbol{\phi}) \mathrm{p}(y; \mu) \\
\propto& \mathrm{p}(\boldsymbol{x}_i, y; \boldsymbol{\phi}, \mu) \\
q_i(y) =& \frac{\mathrm{p}(\boldsymbol{x}_i, y; \boldsymbol{\phi}, \mu)}{\sum_{y'} \mathrm{p}(\boldsymbol{x}_i, y'; \boldsymbol{\phi}, \mu)} \\
=& P(Y_i = y | \boldsymbol{x}_i; \boldsymbol{\theta}, \phi)
\end{aligned}
$$

After normalizing, each $q_i(y)$ – which is the soft distribution over clusters for data $\boldsymbol{x}_i$ – is set to the conditional probability $P(y_i | \boldsymbol{x}_i)$ under the current parameters $\mu, \boldsymbol{\phi}$.

This is called the E-step, or "expectation step," because it is derived from updating the expected likelihood under $q(\boldsymbol{y})$.

The M-step

Next, we hold $q(\boldsymbol{y})$ fixed and update the parameters. Let's do $\boldsymbol{\phi}$, which parametrizes $\mathrm{p}(\boldsymbol{x}|y)$. Again, we start by adding Lagrange multipliers to the lower bound,

$$\mathcal{J} = \sum_i \sum_y q_i(y) \left(\log \mathrm{p}(\boldsymbol{x}_i | Y_i = y; \boldsymbol{\phi}) + \log \mathrm{p}(y; \mu) - \log q_i(y)\right) + \sum_y \lambda_y (1 - \sum_j \phi_{y,j})$$

$$\frac{\partial \mathcal{J}}{\partial \phi_{y,j}} = \sum_i q_i(y) \frac{x_{i,j}}{\phi_{y,j}} - \lambda_y$$

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

$$\phi_{y,j} = \frac{\sum_i q_i(y) x_{i,j}}{\sum_{j'} \sum_i q_i(y) x_{i,j'}} = \frac{E_q[\text{count}(y, j)]}{E_q[\text{count}(y)]}$$

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 Naive Bayes, except we used expected counts rather than observed counts.

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.

- Each step is guaranteed not to decrease $\mathcal{J}$
- This is called hill-climbing: 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.

- But the overall procedure is **not** guaranteed to find a global maximum.
- This means that initialization is important: where you start can determine where you finish.
- This is not true in most of the supervised learning algorithms that we have considered, such as logistic regression; in that case, we are optimizing $\log \mathrm{p}(\boldsymbol{y}|x;\boldsymbol{\theta})$, which is defined so as to be convex with respect to the parameter $\boldsymbol{\theta}$. This means that for logistic regression (and many other supervised learning algorithms), we don't need to worry about initialization, because it won't affect our ultimate solution: we are guaranteed to reach the global minimum.

7.3 Applications of EM

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

- Introduce latent variables $\boldsymbol{z}$, such that it's easy to write the probability $P(\mathcal{D}, \boldsymbol{z})$, where $\mathcal{D}$ is your observed data, and easy to estimate the associated parameters.
- Derive the E-step updates for $q(\boldsymbol{z})$, which is typically factored as $q(\boldsymbol{z}) = \prod_i q_{z_i}(z_i)$.

Some applications of this basic setup are presented here.

Word sense clustering

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

Semi-supervised learning

Nigam et al. (2000) offer another application of EM: **semi-supervised learning**. They apply this idea to document classification in the classic "20 Newsgroup" dataset.

- In this setting, we have labels for some of the instances, $\langle \boldsymbol{x}^{(\ell)}, \boldsymbol{y}^{(\ell)} \rangle$, but not for others, $\langle \boldsymbol{x}^{(u)} \rangle$.

- Can unlabeled data improve learning?

We will choose parameters to maximize the joint likelihood,

$$\log \mathrm{p}(\boldsymbol{x}^{(\ell)}, \boldsymbol{x}^{(u)}, \boldsymbol{y}^{(\ell)}) = \log \mathrm{p}(\boldsymbol{x}^{(\ell)}, \boldsymbol{y}^{(\ell)}) + \log \mathrm{p}(\boldsymbol{x}^{(u)}) \tag{7.4}$$

- We treat the labels of the unlabeled documents as missing data. 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 $q(\boldsymbol{y})$.

- We can further parametrize this approach by weighting the unlabeled documents by a scalar λ, which is a tuning parameter.

Multi-component modeling

- One of the classes in 20 newsgroups is `comp.sys.mac.hardware`.

- Suppose that 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.

We can envision a new generative process here:

- For each document i,
  - draw the label $y_i \sim \text{Categorical}(\theta)$
  - draw the component $z_i | y_i \sim \text{Categorical}(\psi_{y_i})$
  - draw the vector of counts $\boldsymbol{x}_i | z_i \sim \text{Multinomial}(\phi_{z_i})$

Our labeled data includes $\langle \boldsymbol{x}_i, y_i \rangle$, but not z_i, so this is another case of missing data.

$$
\begin{aligned}
\mathrm{p}(\boldsymbol{x}_i, y_i) =& \sum_z \mathrm{p}(\boldsymbol{x}_i, y_i, z) \\
=& \mathrm{p}(\boldsymbol{x}_i | z; \boldsymbol{\phi}) \mathrm{p}(z | y_i; \psi) \mathrm{p}(y_i; \mu)
\end{aligned}
$$

Again, we can apply EM

- We need a distribution over the missing data, $q_i(z)$. This is updated during the E-step.
- During the m-step, we compute:

$$
\begin{aligned}
\psi_{y,z} =& \frac{E_q[\text{count}(y, z)]}{\sum_{z'} E_q[\text{count}(y, z')]} \\
\boldsymbol{\phi}_{j,y,z} =& \frac{E_q[\text{count}(z, j)]}{\sum_{j'} E_q[\text{count}(z, j')]}
\end{aligned}
$$

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

Chapter 8

Language models

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

- machine translation
- speech recognition
- summarization

As we will soon see, we can produce more **fluent** text output by computing the probability of the text.

Specifically, suppose we have a vocabulary of word types

$$\mathcal{V} = \{\textit{aardvark}, \textit{abacus}, \ldots, \textit{zither}\} \tag{8.1}$$

Given a sequence of word tokens $w_1, w_2, \ldots, w_M$, with $w_i \in \mathcal{V}$, we would like to compute the probability $\mathrm{p}(w_1, w_2, \ldots, w_M)$. We will do this in a data-driven way, assuming we have a **corpus** of text.

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

- Language models typically make an independence assumption across sentences, $\mathrm{p}(\boldsymbol{s}_1, \boldsymbol{s}_2, \ldots) = \prod_j \mathrm{p}(\boldsymbol{s}_j)$, where each sentence $\boldsymbol{s}_j = [\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots, \boldsymbol{w}_{N_j}]$.

> So for our purposes, it is sufficient to compute the probability of sentences. The justification for this assumption is that the probability of words that are not in the same sentence don't depend on each other too much. Clearly this isn't true: once I mention *Manuel Noriega* once in a document, I'm far more likely to mention him again (Church, 2000). But the dependencies between words within a sentence are usually even stronger, and are more relevant to the fluency considerations inherent in applications such as translation and speech recognition (which are typically evaluated at the sentence level anyway).

So how can we compute the probability of a sentence? The simplest idea would be to apply a **relative frequency estimator**:

$$\mathrm{p}(\textit{Computers are useless, they can only give you answers}) \tag{8.2}$$

$$= \frac{\mathrm{count}(\textit{Computers are useless, they can only give you answers})}{\mathrm{count}(\text{all sentences ever spoken})} \tag{8.3}$$

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

- In the theoretical limit of infinite data, it might work. But in practice, we are asking for accurate counts over an infinite number of events, since sentences can be arbitrarily long.
- Even if we set an aggressive upper bound of, say, $n = 20$, the number of possible sentences is $\#|\mathcal{V}|^{20}$. A small vocabularly for English would have $\#|\mathcal{V}| = 10^4$, so we would have 10^{80} possible sentences.
- Clearly, this estimator is extremely data-hungry. We need to introduce bias to have a chance of making reliable estimates.

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

- *Colorless green ideas sleep furiously*
- *Furiously sleep ideas green colorless*

Noam Chomsky used this pair of examples to argue that the probability of a sentence is a meaningless concept:

- Any English speaker can tell that the first sentence is grammatical but the second sentence is not.
- Yet neither sentence, nor their substrings, had ever appeared at the time that Chomsky wrote this article (they have appeared lots since then).
- Thus, he argued, empirical probabilities can't distinguish grammatical from ungrammatical sentences.

Pereira (2000) showed that by identifying *classes* of words (e.g., noun, verb, adjective, adverb — but not necessarily these grammatical categories), it is easy to show that the first sentence is more probable than the second. We will talk about class-based language models later.

Are language models useful? Suppose we want to translate a sentence from Spanish:

- *El cafe negro me gusta mucho*.
- Word-for-word: *The coffee black me pleases much*.
- But a good language model of English will tell us:

$$P(\textit{The coffee black me pleases much}) < P(\textit{I like black coffee a lot}) \tag{8.4}$$

- How can we use this fact?

Warren Weaver on translation as decoding:

> 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 motivates a generative model (like Naive Bayes!):

- English sentence $\boldsymbol{w}^{(e)}$ generated from language model $\mathrm{p}_e(\boldsymbol{w}^{(e)})$
- Spanish sentence $\boldsymbol{w}^{(s)}$ generated from noisy channel $\mathrm{p}_{s|e}(\boldsymbol{w}^{(s)}|\boldsymbol{w}^{(e)})$

(picture)
Then the **decoding** problem is: $\max_{\boldsymbol{w}^{(e)}} \mathrm{p}(\boldsymbol{w}^{(e)}|\boldsymbol{w}^{(s)}) \propto \mathrm{p}(\boldsymbol{w}^{(s)}, \boldsymbol{w}^{(e)}) = \mathrm{p}(\boldsymbol{w}^{(e)})\mathrm{p}(\boldsymbol{w}^{(s)}|\boldsymbol{w}^{(e)})$

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

What else can we model with a noisy channel?

- Speech recognition (original = words; encoded = sound)
- Spelling correction (original = well-spelled text; encoded = text with spelling mistakes)
- Part of speech tagging (original = tags; encoded = words)
- Parsing (original = parse tree; encoded = words)
- ...

The noisy channel model allows us to decompose NLP systems into two parts:

- The translation model, which we need labeled data to estimate.
- The language model, which we need only *unlabeled* data to estimate.

Since there is always more unlabeled data, this means we can improve NLP systems just by improving $\mathrm{p}_e(\boldsymbol{w})$.

8.1 N-gram language models

We began with the relative frequency estimator,

$$\mathrm{p}(\textit{Computers are useless, they can only give you answers}) \tag{8.5}$$

$$= \frac{\mathrm{count}(\textit{Computers are useless, they can only give you answers})}{\mathrm{count}(\text{all sentences ever spoken})} \tag{8.6}$$

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

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

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

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

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

N-gram models

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

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

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

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

- To compute the probability of a whole sentence, it's convenient to pad the beginning and end with special symbols $\Diamond$ and $\Box$. Then the bigram ($n=2$) approximation to the probability of *I like black coffee* is:

$$\mathrm{p}(\textit{I} \mid \Diamond)\mathrm{p}(\textit{like} \mid \textit{I})\mathrm{p}(\textit{black} \mid \textit{like})\mathrm{p}(\textit{coffee} \mid \textit{black})\mathrm{p}(\Box \mid \textit{coffee}) \tag{8.7}$$

- In this model, we have to estimate and store the probability of only $\#|\mathcal{V}|^n$ events. A very common choice is a trigram model, in which $n=3$.
- The n-gram probabilities can be determined by relative frequency estimation,

$$\mathrm{p}(w|u,v) = \frac{\text{count}(u,v,w)}{\text{count}(u,v)} = \frac{\text{count}(u,v,w)}{\sum_{w'} \text{count}(u,v,w')} \tag{8.8}$$

There could be too problems with an n-gram language model:

- n **is too small**. In this case, we are missing important linguistic context. Consider the following sentences:
  - Gorillas *always like to groom* ***THEIR*** *friends.*
  - *The* computer *that's on the 3rd floor of our office building* ***CRASHED****.*

  The bolded words depend crucially on their predecessors in italics: *their* depends on knowing that *gorillas* is plural, and *crashed* depends on knowing that the subject is a *computer*. The resulting model would offer probabilities that are too low for these sentences, and too high for sentences that fail basic linguistic tests like number agreement.

- n **is too big**. In this case, we can't make good estimates of the n-gram parameters from our dataset. See the slides for some examples of this.

- These two problems point to another **bias/variance** tradeoff. Can you see how it works?

- In reality, we often have **both** problems! Language is full of long-range dependencies, and datasets are small.

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

8.2 Evaluating language models

- Because language models are typically components of larger systems (language modeling is not really an application itself), we would prefer **extrinsic evaluation**: does the LM help the task (translation or whatever). But this is often hard to do, and depends on details of the overall system which may be irrelevant to language modeling.

- **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).

Held-out likelihood

A popular intrinsic metric is the **held-out likelihood**.

- We obtain a test corpus, and compute the (log) probability according to our model. It is crucial that the words in this corpus were not used in estimating the model itself.
- A good model should assign high probability to this held-out data.
- Specifically, we compute

$$\ell(\boldsymbol{w}) = \sum_i \sum_m \log \mathrm{p}(w_m^{(i)} | w_{m-1}^{(i)}, \ldots, w_{m-n+1}^{(i)}), \tag{8.9}$$

  for all sentences $\boldsymbol{w}^{(i)}$ in the held-out corpus.

Perplexity

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

$$PP(\boldsymbol{w}) = 2^{-\frac{\ell(\boldsymbol{w})}{M}}, \tag{8.10}$$

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

- After this transformation, we now prefer lower values. In the limit, we obtain probability 1 for our held-out corpus, with $PP = 2^{-\log 1} = 1$.
- Assume a uniform, unigram model in which $P(s_i) = \frac{1}{V}$ for all V words in the vocabulary. Then,

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

- We can think of perplexity as the *weighted branching factor* at each word in the sentence.
  - If we have solved the word prediction problem perfectly, $PP(\boldsymbol{w}) = 1$, because there is only one possible choice.

- If we have only a uniform model that assigns equal probability to every word, $PP(\boldsymbol{w}) = V$.
- Most models fall somewhere in between.
- Here's how you remember: lower perplexity is better, because you are less perplexed.

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

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

Will it keep going down? See slides from (Manning and Schütze, 1999).

Information theory*

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

$$H(P) = -\sum_{\boldsymbol{w}} \mathrm{p}(\boldsymbol{w}) \log \mathrm{p}(\boldsymbol{w}) \tag{8.11}$$

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

Cross-entropy is a relationship between two probability distributions, the true one $P(W)$ and an estimate $Q(W)$.

$$\begin{aligned}
H(P, Q) =& E_P[\log Q] \\
=& -\sum_{\boldsymbol{w}} \mathrm{p}(\boldsymbol{w}) \log q(\boldsymbol{w}) \\
=& \sum_{\boldsymbol{w}} \mathrm{p}(\boldsymbol{w}) \log \frac{\mathrm{p}(\boldsymbol{w})}{q(\boldsymbol{w})} - \mathrm{p}(\boldsymbol{w}) \log \mathrm{p}(\boldsymbol{w}) \\
=& D_{KL}(P||Q) + H(Q)
\end{aligned}$$

So the cross-entropy is the KL-divergence between P and Q – a non-symmetric distance measure between distributions, which we will see again later in the course – plus the entropy of P. Since P is the language itself, we can only control Q, and minimizing the cross-entropy is equivalent to minimizing the KL-divergence.

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

$$\begin{aligned}
H(P, Q) =& -\sum_{\boldsymbol{w}} \mathrm{p}(\boldsymbol{w}) \log q(vw) \\
=& -\lim_{M \to \infty} \frac{1}{M} \log q(\boldsymbol{w}) \\
\approx& -\frac{1}{M} \log q(\boldsymbol{w}) \\
PP(S) =& 2^{-\frac{1}{M} \log q(vw)}
\end{aligned}$$

A good language model has low cross-entropy with $P(W)$, and thus low perplexity.

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

- Speakers shorten predictable words, lengthen unpredictable ones
- High-entropy sentences take longer to read
- Syntactic reductions (e.g., *I'm* versus *I am*) are more likely when the reducible word contains less information.

8.3 Smoothing and discounting

We want to estimate $P(W)$ from sparse statistics, avoiding $\mathrm{p}(\boldsymbol{w}) = 0$.

Laplace/Lidstone smoothing

Simplest idea: just add "pseudo-counts"

$$\mathrm{p}_{\text{Laplace}}(w \mid v) = \frac{\text{count}(v, w) + \alpha}{\sum_{w'} \text{count}(v, w') + V\alpha} \tag{8.12}$$

Anything that we add to the numerator (α) must also appear in the dominator ($V\alpha$). We can capture this with the concept of **effective counts**:

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

The **discount** for each n-gram is:

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

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

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.
- Instead, we could borrow the same amount of probability mass from all observed counts, and redistribute it among only the unobserved counts. This is called **absolute discounting.**

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

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

- We need not redistribute the probability mass equally. Instead, we can **backoff** to a lower-order language model.
- In other words: if you have trigrams, use trigrams; if you don't have trigrams, use bigrams; if you don't even have bigrams, use unigrams. (And what if you don't even have unigrams?). This is called **Katz backoff**.

$$c^*(u,v) = c(u,v) - d$$

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

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

Interpolation

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

Instead of choosing a single n-gram order, we can take the weighted average:

$$\begin{aligned} \mathrm{p}_{\text{Interpolation}}(w|u,v) = {} & \lambda_1 \mathrm{p}_1^*(w|u,v) \\ & + \lambda_2 \mathrm{p}_2^*(w|u) \\ & + \lambda_3 \mathrm{p}_1^*(w) \end{aligned}$$

- p^*_k is the maximum likelihood estimate (MLE) of a k-gram model
- Constraint: $\sum_z \lambda_z = 1$
- We can tune λ on heldout data...
- Or we can use **expectation maximization**!

EM for interpolation We can add a latent variable z_m, indicating the order of the n-gram that generated word w_m. Generative story:

- For each word m
  - Draw $z_m \sim \text{Categorical}(\lambda(w_m))$
  - Draw $w_m \sim \mathrm{p}^*_{z_m}(w_m | s_{m-1}, \ldots s_{m-z_m})$

As always we have two quantities of interest in our EM application:

- The parameters, λ.
- Our beliefs about the latent variables. Let $q_m(z)$ be our degree of belief that word token w_m was generated from a n-gram of order z.

Having defined these quantities, we can derive EM updates:

- **E-step**: $q_m(z) = \mathrm{p}(z | w_{1:m}) = \frac{\mathrm{p}^*_z(w_m | w_{m-1}, \ldots, w_{m-z+1})}{\sum_{z'} \mathrm{p}^*_{z'}(w_m | w_{m-1}, \ldots w_{m-z'+1})} \mathrm{p}(z' | \lambda(w_m))$
- **M-step**: $\lambda(w)_z = \frac{E_q[\text{count}(W=w, Z=z)]}{\sum_{z'} E_q[\text{count}(W=w, Z=z')]}$

By running the EM algorithm, we can obtain a good estimate of λ, which we can then use for unseen data. It should be clear how we can extend this approach to trigrams and beyond; Collins (2013) offers more details.

Kneser-ney smoothing

Kneser-ney smoothing also incorporates discounting, but redistributes the resulting probability mass in a different way. Consider the example:

I recently visited

- *Francisco*?
- *Duluth*?

Key idea: some words are more **versatile** than others.

- Suppose $\mathrm{p}^*(\textit{Francisco}) > \mathrm{p}^*(\textit{Duluth})$, and $c(\textit{visited Francisco}) = c(\textit{visited Duluth}) = 0$.
- We would still guess that $\mathrm{p}(\textit{visited Duluth}) > P(\textit{visited Francisco})$, because *Duluth* is a more versatile word.

We define the Kneser-Ney bigram probability as

$$\mathrm{p}_{KN}(v|u) = \begin{cases} \frac{\text{count}(u,v)-d}{\text{count}(u)}, & \text{count}(u,v) > 0 \\ \alpha(u)\mathrm{p}_{\text{continuation}}(v), & \text{otherwise} \end{cases}$$
$$\mathrm{p}_{\text{continuation}}(v) = \frac{\#|u : \text{count}(u,v) > 0|}{\sum_{v'} \#|u' : \text{count}(u',v') > 0|}$$

- We reserve probability mass using absolute discounting d.
- The *continuation probability* $\mathrm{p}_{\text{continuation}}(u)$ is proportional to the number of observed contexts in which u appears.
- As in Katz backoff, $\alpha(v)$ makes the probabilities sum to 1
- In practice, interpolation works a little better than backoff

$$\mathrm{p}_{KN}(v|u) = \frac{\text{count}(u,v) - d}{\text{count}(u)} + \lambda(u)\mathrm{p}_{\text{continuation}}(v) \tag{8.13}$$

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

8.4 Other types of Language Models

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

Mixed-order n-gram models

Saul and Pereira (1997) described a "mixed-order" n-gram model, where you condition on multiple bigram contexts, skipping over intermediate words:

$$\mathrm{p}(w_m|w_{m-1}, \ldots, w_{m-n+1}) = \sum_k \lambda_k(w_{m-k})\tilde{\mathrm{p}}(w_m|w_m - k)\prod_j^{k-1}[1 - \lambda_j(w_{m-j})] \quad (8.14)$$

- This is an **interpolated** model, because we are taking the weighted average over a bunch of bigram probabilities.
- Note that the interpolation weight depends on the context word, $\lambda_k(w_{m-k})$. This means that some words can prefer certain dependency lengths — for example, adjectives might prefer short dependencies, since they tend to affect adjacent nouns, while verbs might prefer longer dependencies, since they can affect indirect objects that are further away.
- The final product ensures that the weights in any particular context must add up to one: each λ_k is taking a slice of the probability mass that has already been used by the earlier contexts $j < k$.
- The parameters $\lambda_k(w)$ can be estimated by expectation maximization, just like in the interpolated N-gram model above.

Class-based language models

The reason we need smoothing is because the trigram probability model $\mathrm{p}(w|u, v)$ has a huge number of parameters. Let's simplify:

$$\mathrm{p}_{\text{class}}(w|v) = \sum_z P(w|z;\theta)P(z|v;\phi),$$

where $z \in [1, K], K \ll V$.

We get a bigram probability using $2VK$ parameters instead of V^2.

We could use EM to estimate θ and ϕ (Saul and Pereira, 1997).

- The latent variable is the class z, so the e-step updates $q_m(z)$
- The parameters are θ and ϕ, which can be updated in the M-step.

But this is usually too slow, so there are approximate algorithms, like "exchange clustering" (Brown et al 1992), which assigns each word type to a single class.

Discriminative language models

- Or we could just train a model to predict $\mathrm{p}(w_m|w_{m-1}, w_{m-2}, \ldots)$ directly.
- We might be able to use arbitrary features of the history to model long-range dependencies.
- Algorithms such as perceptron and logistic regression have been considered (Rosenfeld, 1996; Roark et al., 2007)
- Currently, "neural probabilistic language models" are attracting a lot of interest. The log-bilinear model (Mnih and Hinton, 2008) looks like this:

$$\mathrm{p}^h_\theta(w) = \frac{\exp(s_\theta(w,h))}{\sum_{w'} \exp(s_\theta(w',h))}$$
$$s_\theta(w,h) = \hat{\boldsymbol{q}}_h^\top \boldsymbol{q}_w + b_w,$$

  where h is the history context, $\hat{\boldsymbol{q}}_h$ is a latent description of the history, $\boldsymbol{q}_w$ is a latent description of the word, and b_w is an offset. The history context can be computed from the words themselves, as $\hat{\boldsymbol{q}}_h = \sum_i^{m-1} C_i \boldsymbol{q}_i$, where the matrix C_i is applied to context position i. All parameters can be estimated to directly maximize the probability of a corpus, using gradient ascent.

- Recent work has focused on efficiently training such models, with increasingly convincing results on large training sets (Mikolov et al., 2011).

Chapter 9

Morphology

[1] So far we have been focusing on NLP at the word level. Today we go **inside of words**.

We've already hinted at a morphological problem by introducing the idea of **lemmas**, where *serve/served/serving* all have the lemma *serve*.

From the perspective of document classification, these multiple forms may just seem like an annoyance, which we can get rid of by lemmatization or stemming (more on this later).

But morphology conveys information which might be crucial for some applications:

- Information retrieval
  - With a query like *bagel*, we want to get hits for *bagels*.
  - Same for *corpus*/*corpora*, *goose*/*geese*.
  - But we don't always want all the inflected forms. For example, a query for *Apple* may not want hits for *apples*
- Time. Morphology often indicates when events happen. For example, in French:

J'achete un velo	I buy a bicycle (now)
J'acheterai un velo	I will buy a bicycle
J'achetais un velo	I was buying a bicycle
J'ai acheté un velo	I bought a bicycle
J'acheterais un velo	I would buy a bicycle

[1] This chapter is pretty rough; better to see Chapter 2 of (Bender, 2013).

- Causality. Consider the difference between the Spanish examples:

*Si tu vas a GT, tu **seras** rica*	If you go to GT, you will be rich
*Si tu vas a GT, tu **eres** rica.*	If you go to GT, you are rich

- Lexical semantics: suppose *antichrist* is not in your sentiment dictionary. Do you think it is a positive or negative word?

In addition to recognizing morphology, there are applications in which we need to produce it.

- Translation: *you (pl) are smart* → *Ustedes son inteligentes* vs *Tu eres inteligente*
- Text generation: `(has-property you-pl smart)` → *ustedes son inteligentes*

Morphology, Orthography, and Phonology

- **Morphology** describes how meaning is constructed from combining affixes. For example, it is a morphological fact of English that adding the affix *+S* to many nouns creates a plural.

$$\textit{berry} + \textsc{Plural} \rightarrow \textit{berry+s}$$

- **Orthography** specifically relates to writing.
For example,

$$\textit{berry+s} \rightarrow \textit{berries}$$

is an orthographic rule. We have lots of these in English, which is one reason English spelling is difficult.

- Morphological rules also include stem changes, such as *goose* + PLURAL → *geese*.
- **Phonology** describes how sounds combine. For example, the different pronunciations of the final *s* in *cats* (s) and *dogs* (z) follow from a phonological rule (example (25) in the Bender text, page 30).
- In English, morphologically distinct words may be pronounced differently even when they are spelled the same, and this can reflect morphological differences. *read*+PRESENT vs. *read*+PAST.
- Conversely, morphological variants may be spelled differently even when they sound the same, like *The Champions' league* vs *The Champion's league* vs *The Champions league*.

Productivity

One idea for dealing with morphology is to build a morphologically aware dictionary:

- Map each **surface form** to its underlying **lemma**
- Include meaning of morphology: tense, number, animacy, posession, etc.
- Then when you encounter a surface form, just look it up.

duck	*duck*/N+SG
ducks	*duck*/N+PL
duck	*duck*/VB+PRESENT
ducks	*duck*/VBZ+PRESENT

Will this work? Besides the problem of ambiguity, still another problem is that morphology is **productive**, meaning that it applies to new words. If you only know the words *Google* or *iPad*, you can immediately understand their inflected forms.

- Have you Googled that yet?
- I have owned three iPads.

Derivational morphology (more on this later) is productive in another way: you can produce new words by applying morphological changes to existing words. hyper+un+desire+able+ity

In some languages, derivational morphology can create extremely complicated words. The J&M textbook has a fun example from Turkish:

In the homework, you'll see examples from Swahili, which also has complex morphology. A dictionary of all possible surface forms in such languages would be gargantuan. So instead of building a static dictionary, we will try to model the underlying morphological and orthographic rules.

9.1 Morphemes

Two broad classes: **stems** and **affixes**.

- Intuitively, stems are the "main" part of meaning, affixes are the modifiers

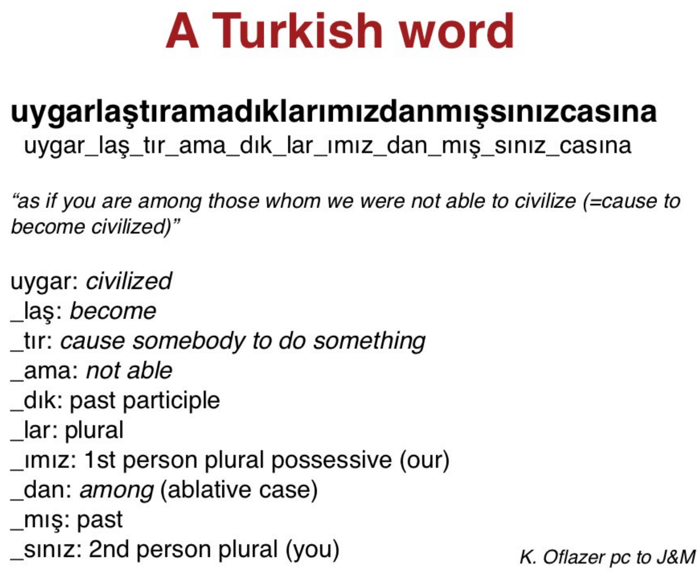

Figure 9.1: From (Jurafsky and Martin, 2009)

- Typically, **stems** can appear on their own (they are **free**) and affixes cannot (they are **bound**).
- Types of affixes:
  - **Prefixes**: *un+learn*, *pre+view*.
    - These examples are derivational. English has few inflectional prefixes, but other languages have many.
    - For example, in Swahili: *u-na-kata* versus *u-me-kata* distinguishes *you are cutting* versus *you have cut*. *na* and *me* are prefixes, *kata* is the root.
  - **Suffixes**: *I learn+ed*, *She learn+s*, *three apple+s*, *four fox+es*.
  - **Circumfixes** go around the stem.
    - None in English.
    - German has a circumfix for the past participle: *sagen* (say) → *ge+sag+t* (said)
    - French negation can be seen as a circumfix: *Je mange+NEG*→*Je ne mange pas*. (I do not eat).

* More generally, morphemes can be non-continuous, as in the Hebrew example (7) in the Bender reading (page 12).

(7)

Root	Pattern	Part of Speech	Phonological Form	Orthographic Form	Gloss
ktb	CaCaC	(v)	katav	כתב	'wrote'
ktb	hiCCiC	(v)	hixtiv	הכתיב	'dictated'
ktb	miCCaC	(n)	mixtav	מכתב	'a letter'
ktb	CCaC	(n)	ktav	כתב	'writing, alphabet'

[heb]

In this example, the root *ktb* (related to writing) is combined with patterns that indicate where to insert vowels to produce different parts-of-speech and meanings.

– **Infixes** go inside the stem.

* Tagalog: *hingi+AGENT→h+um+ingi*
* Lakhota: *m'ani* (he walks), *ma-wá-ni* (I walk). The *wá* marks agreement with a first-person singular subject; it is an infix for this word, although it is a prefix in other words.
* English: *absolutely+fucking→absofuckinglutely*, but **absfuckingsolutely* arguably doesn't work.

9.2 Types of morphology

- **Inflection** creates different forms of a single word:
  - tense: *to be, being, I am, you are, he is, I was*
  - number: *book, books*
  - case: *he, his, her, they, them, their*
- **Derivation** creates new words:
  grace → disgrace → disgraceful → disgracefully
- **Cliticization** combines *Georgia+'s* into *Georgia's*; the possessive clitic *'s* is syntactically independent but phonologically dependent.

- Pronouns appear as clitics in French, e.g., *j'accuse* (I accuse), as does negation *Je n'accuse personne* (I don't accuse anyone).
- Another example is from Hebrew: *l'shana tova* (literally for year good, meaning happy new year); the preposition *for* appears as a clitic.

- **Compounding** combines two words in a new word:
  cream $\rightarrow$ *ice cream* $\rightarrow$ *ice cream cone* $\rightarrow$ *ice cream cone bakery*

- **Portmanteaus** combine words, truncating one or both.
  smoke + *fog* $\rightarrow$ *smog*
  glass + *asshole* $\rightarrow$ *glasshole*

- Word formation is *productive*: new words are subject to all of these processes

Inflectional morphology

Inflectional morphology adds information about words. English has a very simple system of inflectional morphology, compared to many languages.

Affix	Syntactic/semantic effect	Examples
-s	NUMBER: plural	*cats*
-'s	possessive	*cat's*
-s	TENSE: present, SUBJ: 3sg	*jumps*
-ed	TENSE: past	*jumped*
-ed/-en	ASPECT: perfective	*eaten*
-ing	ASPECT: progressive	*jumping*
-er	comparative	*smaller*
-est	superlative	*smallest*

Figure 9.2: From (Bender, 2013)

- English nouns are marked for number and possession; many language also mark nouns for **case**, which is the syntactic role that the noun plays in the sentence.

- In English, we do distinguish the case of some pronouns:
  * *He* (NOMINATIVE) *gave her* (OBLIQUE) *his* (GENITIVE) *guitar.*
  * *She gave him her guitar.*
  * *I gave you our guitar.*
  * *You gave me your guitar.*

  Specifically, we distinguish the **nominative** case of personal pronouns (except for 2nd-person), and the **genitive** case; all other uses are the **oblique** case.
- Other languages – such as Latin, Russian, Sanskrit, and Tamil, mark the case of all nouns. These languages have additional cases, such as dative (indirect object), accusative (direct object), and vocative (address).
- In German, noun is not inflected for case, but the articles and adjectives are:
  * *Der alte Mann gab dem kleinen Affen die grosse Banane* (based on example 49 from Bender)
  * The old man (NOM) gave the little monkey (DATIVE) the big banana (ACCUSATIVE)
  * Notice how *der*, *dem*, and *die* all mean the, but carry the case marking.
  * The adjectives are also marked for case.
- Many languages – such as Romance languages – mark the gender and number of nouns by inflecting the article and adjective. e.g., Spanish:
  * *El coche rojo pasó la luz roja*: the red car ran the red light
  * *Los coches rojos pasó las luces rojas*: the red cars ran the red lights
  * Note that the article and adjective must **agree** for the sentence to be grammatical.
  * In English, demonstrative determiners mark number, *this book* vs *these books*. In English, the determiner and noun must agree, e.g. **this books*.
- Gender is not necessarily binary.
  * English pronouns include neuter *it*; German, Sanskrit, and Latin do this for all nouns.
  * Danish and Dutch distinguish **neuter** from **common** gender
  * Other languages distinguish **animate** and **inanimate**

  - Number if not necessarily binary.
    - Many languages, such as Arabic and Sanskrit, include a special **dual** number for two. English has residual traces of the dual number, with *both* vs *all* and *either* vs *any*.
    - Some Austronesian languages have a **trial** number, for groups of three.
    - Some languages, including Arabic, have a **paucal** number, for small groups.

- English verbs are inflected for tense and number distinguishing past (*I ate*), present (*I eat*), and 3rd-person singular (*She eats*). They are also inflected for aspect, distinguishing perfective (*I had eaten*) and progressive (*I am eating*). Note that the perfective and the past tense are identical for regular verbs, e.g. *we had talked*, *we talked*.

  - Many languages (e.g., Chinese and Indonesian), do not mark tense with morphology. Indonesian uses time signals.

Saya makan apel	I eat an apple
Saya sedang makan apel	I am eating an apple
Saya telah makan apel	I already ate an apple
Saya akan makan apel	I will eat an apple

  - Romance languages distinguish many more tenses than English with morphology.
    - Spanish has multiple past tenses: **preterite** and **imperfect**, distinguishing, e.g. *I ate onions yesterday* from *I ate onions every day*. These are distinguished by morphology: *comí cebollas ayer*, *comía cebollas cada día*.
    - Spanish and French have endings for conditional (*comería cebollas*) and future (*comeré cebollas*)
    - All of these are marked with time signals in English; future can also be marked this way in French and Spanish, e.g. *voy a comer cebollas*.
  - Romance also have separate forms for every combination of number and person.
    - *(yo hablo / tu hablas / ella habla / nosotros hablamos / vosotros hableis / ellas hablan)*
    - *(je parle / tu parles / elle parle / nous parlons / vous parlez / ils parlent)*

  * In Spanish, they they eliminate pronouns (pro-drop) in cases where the morphology makes it clear (unless they want to add emphasis). Chinese is also a pro-drop language?
  * This doesn't happen in French, maybe because many different spellings (*parle/parles/parlent*) sound the same.
  * In English, we only distiniguish 3rd-person singular.

- Adjectives in English mark comparative and superlative (*taller*, *tallest*). As we have seen, they can mark gender and number in other languages.

- Other things can be marked with affixes, such as **evidentiality** – how the speaker came to know the information. In Eastern Pomo (a California language), there are verb suffixes for four evidential categories:

-ink'e	nonvisual sensory
-ine	inferential
-le	hearsay
-ya	direct knowledge

  Example (41) in the text shows evidentiary marking in Turkish, *Ahmet geldi* (Ahmet came, witnessed by the speaker) vs *Ahmet gelmi s* (not witnessed by the speaker)

The **index of synthesis** measures the ratio of the number of morphemes in a given text to the number of words. Languages with complex morphology are called **synthetic**; languages with simple morphology are called **isolating** or **analytic**. English is relatively, but not extremely, analytic.

Language	Index of synthesis
Vietnamese	1.06
Yoruba	1.09
English	1.68
Old English	2.12
Swahili	2.55
Turkish	2.86
Russian	3.33
Inuit (Eskimo)	3.72

Figure 9.3: From (Bender, 2013)

An approximation of the index of synthesis is the type-token ratio. Can you see why? If you count the number of unique surface forms in 10K *parallel* sentences from Europarl, you get:

- English: 16k word types
- French: 22k
- German: 32k
- Finnish: 55k

Derivational Morphology

Derivational morphology is a way to create new words and change part-of-speech.

- **nominalization**
  - *V + -ation: computerization*
  - *V + -er: walker*
  - *Adj + -ness: fussiness*
  - *Adj + -ity: obesity*
- **negation**: *undo, unseen, misnomer*
- **adjectivization**: *V + -able : doable, thinkable*, *N + -al : tonal, national*, *N + -ous: famous, glamorous*
- **abverbization**: *ADJ + -ily: clumsily*
- **lots more**: *rewrite*, *phallocentrism*, *...*

You can create totally new words this way.
word $\rightarrow$ *wordify* $\rightarrow$ *wordification* $\rightarrow$ *wordificationism* $\rightarrow$ *antiwordificiationism* $\rightarrow$ *hyperantiwordificationism*

Irregularities

English morphology contains a lot of irregularities: *know/knew/known*, *foot/feet*, *go/went*.. if you're not a native speaker, learning these was probably a pain in the neck.

- the good news is, there are fewer of these all the time! for example, the past tense of *show* used to be *shew*, like *know/knew* (the past participle is still *shown*).
- the bad news is, the most common words will be the last to change (if ever).

Attaching affixes can cause orthographic and phonological changes:

- walk + ed = walked, but frame+ed = framed, emit+ed = emitted, easy + ier = easier
- this is usually due to phonetic or orthographic *constraints*
- *optimality theory* is an approach to systematizing such interacting constraints. There's a lot of research on finite state models of optimality theory, but you'll have to take a linguistics course for that Karttunen and Beesley (2005).

Chapter 10

Finite-state automata

Finite-state automata are a powerful formalism for representing a subset of formal languages, the **regular** languages. As we will see, this formalism can also be used as a building block for an incredibly wide range of methods for manipulating natural language too (Mohri et al., 2002). This chapter will especially focus on **morphology**, which concerns how words are built out of smaller units. For a good reference on morphology for natural language processing, see (Bender, 2013).

Knight and May (2009) show how finite-state automata can be composed together to create impressive applications. They start with one such application — transliteration — and explain how it works. 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.

10.1 Automata and languages

Basics of the formalism :

- An alphabet Σ is a set of symbols
- A string ω is a sequence of symbols.
  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 an input string, and either accepts or rejects it.

Chomsky Hierarchy Every automaton defines a language. Different automata define different classes of languages. The Chomsky Hierarchy:

- Finite-state automata define **regular** languages
- Pushdown automata define **context-free** languages
- Turing machines define **recursively-enumerable** languages

Finite-state automata A finite-state automaton $M = \langle Q, \Sigma, q_0, F, \delta \rangle$ consists of:

- A finite set of states $Q = \{q_0, q_1, \ldots, q_n\}$
- A finite alphabet Σ of input symbols
- A start state $q_0 \in Q$
- A set of final states $F \subseteq Q$
- A transition function δ

Determinism

- In a deterministic (D)FSA, $\delta : Q \times \Sigma \to Q$.
- In a nondeterministic (N)FSA, $\delta : Q \times \Sigma \to 2^Q$
- We can determinize any NFSA using the powerset construction, but the number of states in the resulting DFSA may be 2^n.
- 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 (that is, transitions which 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 We've now talked about three operations: union, determinization and minimization. Other important operations are:

intersection : only accept strings in both FSAs

negation only accept strings not accepted by FSA M

concatenation . accept strings of the form $s = [s_1 s_2]$, where $s_1 \in M_1$ and $s_2 \in M_2$

FSAs are closed under all these operations, meaning that resulting automaton is still an FSA (and therefore still defines a regular language).

10.2 FSAs for Morphology

Now for some morphology. Suppose that we want to write a program that accepts words that could **possibly** be constructed in accordance with English derivational morphology, but none of the impossible ones:

- *grace, graceful, gracefully*
- *disgrace, disgraceful, disgracefully, ...*
- *Google,Googler,Googleology,...*
- **gracelyful*, **disungracefully*, ...

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*.
Let's start with some examples:

- *grace*: $q_0 \rightarrow_{\text{stem}} q_1$

- *dis-grace*: $q_0 \rightarrow_{\text{prefix}} q_1 \rightarrow_{\text{stem}} q_2$
- *grace-ful*: $q_0 \rightarrow_{\text{stem}} q_1 \rightarrow_{\text{suffix}} q_2$
- *dis-grace-ful*: $q_0 \rightarrow_{\text{prefix}} q_1 \rightarrow_{\text{stem}} q_2 \rightarrow_{\text{suffix}} q_3$

Can we generalize these examples?

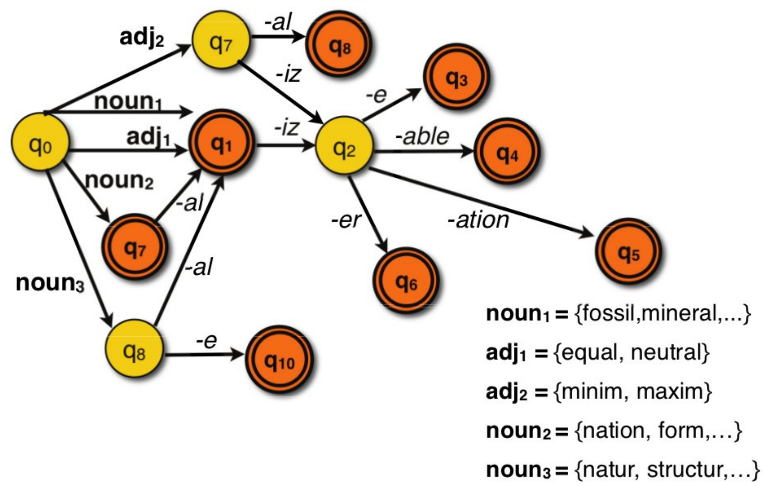

Figure 10.1: I can't find the attribution for this figure right now, sorry! I think it's either from Julia Hockenmaier's slides, or from Jurafsky and Martin (2009).

- This example abstracts away important details, like why *wordificate* is preferred to **wordifycate*. But this rule is part of English **orthography** (spelling), not **morphology**. "Two-level morphology" is an approach to integrating such orthographic transformations in a finite-state framework (Karttunen and Beesley, 2001).
- It also misses a key point: sometimes we have choices, and not all choices are considered to be equally good by fluent speakers.
  - Google counts:
    - *superfast*: 70M; *ultrafast*: 16M; *hyperfast*: 350K; *megafast*: 87K
    - *suckitude*: 426K; *suckiness*: 378K
    - *nonobvious*: 1.1M; *unobvious*: 826K; *disobvious*: 5K

  - 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'll need to augment the formalism for this.

10.3 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 added a weight function that scores every possible transition.

- We can score any path through the WFSA by the sum of the weights.
- Arcs that we don't draw have infinite cost.
- The shortest-path algorithm can find the minimum-cost path for accepting a given string in $O(V \log V + E)$.

Applications of WFSAs

We can use WFSAs to score derivational morphology as suggested above. But let's start with a simpler example:

Edit distance . We can build an edit distance machine for any word. Here's one way to do this (there are others):

- Charge 0 for "correct" symbols and rightward moves
- Charge 1 for self-transitions (insertions)

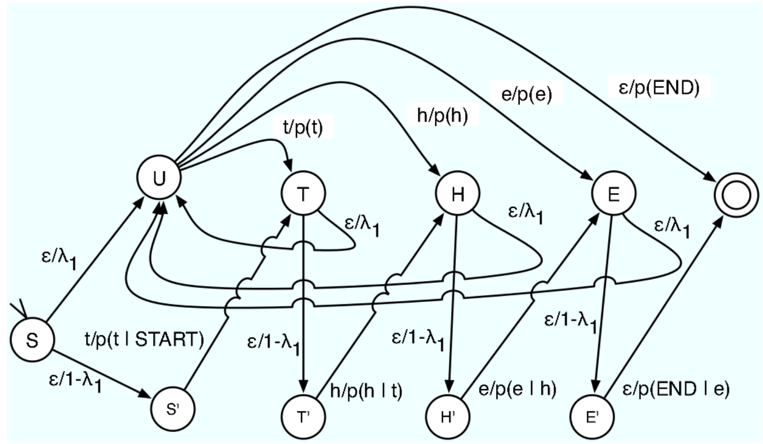

Figure 10.2: From (Knight and May, 2009)

- Charge 1 for rightward epsilon transitions (deletions)
- Charge 2 for "incorrect" symbols and rightward moves (substitutions)
- Charge ∞ for everything else

The total edit distance is the *sum* of costs across the best path through machine.

Probabilistic models For probabilistic models, we make the path costs equal to the likelihood:

$$\delta(q_1, s, q_2) = \mathrm{p}(s, q_2 \mid q_1) \tag{10.1}$$

This enables probabilistic models, such as N-gram language models.

- A unigram language model is just one state, with V edges.
- A bigram language model will have V states, with V^2 edges.

Knight and May (2009) show how to do an interpolated bigram/unigram language model using a WFSA. (Last year I wrote a note that I had found a better way, with only $V + 3$ states rather than $2V + 4$. But now I can't find my solution!)

- Recall that an interpolated bigram language model is

$$\hat{\mathrm{p}}(v|u) = \lambda \mathrm{p}_2(v|u) + (1 - \lambda)\mathrm{p}_1(v), \tag{10.2}$$

with $\hat{\mathrm{p}}$ indicating the interpolated probability, p_2 indicating the bigram probability, and p_1 indicating the unigram probability.

- Unlike the basic n-gram language models, our interpolated model has non-determinism: do we choose the bigram context or the unigram context?
- What should happen to the scores as we encounter a non-deterministic choice?
- For a sequence *a,b,a*, we want the final path score to be

$$\begin{aligned}\psi(a,b,a) =& (\lambda \mathrm{p}_2(a|*) + (1-\lambda)\mathrm{p}_1(a)) \\ & \times (\lambda \mathrm{p}_2(b|a) + (1-\lambda)\mathrm{p}_1(a)) \\ & \times (\lambda P_2(b|a) + (1-\lambda)P(b))\end{aligned}$$

- So we could multiply along each step, and add probabilities across non-deterministic choices.
- With log-probabilities, we would add along each step, and use the log sum, $\log(e^a + e^b)$, to compute the score for non-deterministic branchings.

10.4 Semirings

We have now seen three examples: an acceptor for derivational morphology, and weighted acceptors 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.

- How can we combine all of these possibilities into a single formalism? The answer is semiring notation.

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	$\mathbb{R} \cup -\infty \cup \infty$	$\oplus_{\log}$	$+$	$-\infty$	0	log marginal probability
Tropical	$\mathbb{R} \cup -\infty \cup \infty$	min	$+$	∞	0	best single path

where $\oplus_{\log}(a, b)$ is defined as $\log(e^a + e^b)$.

Semirings allow us to compute a more general notion of the "shortest path" for a WFSA.

- Our initial score is $\overline{1}$
- When we take a step, we use $\otimes$ to combine the score for the step with the running total.
- When nondeterminism lets us take multiple possible steps, we combine their scores using $\oplus$.

Example Let's see how this works out for our language model example.

$$\begin{aligned}
score(\{a, b, a\}) = \overline{1} &\otimes (\lambda \otimes P_2(a|*) \oplus (1-\lambda) \otimes P_1(a)) \\
&\otimes (\lambda \otimes P_2(b|a) \oplus (1-\lambda) \otimes P_1(b)) \\
&\otimes (\lambda \otimes P_2(a|b) \oplus (1-\lambda) \otimes P_1(a))
\end{aligned}$$

Now if we plug in the **probability semiring**, we get

$$\begin{aligned}
score(\{a, b, a\}) = 1 &\times (\lambda P_2(a|*) + (1-\lambda) P_1(a)) \\
&\times (\lambda P_2(b|a) + (1-\lambda) P_1(b)) \\
&\times (\lambda P_2(a|b) + (1-\lambda) P_1(a))
\end{aligned}$$

But if we plug in the **log probability semiring**, we get

$$\begin{aligned}
score(\{a, b, a\}) = 0 &+ \log\left(\exp(\lambda + \log P_2(a|*)) + \exp((1-\lambda) + \log P_1(a))\right) \\
&+ \log\left(\exp(\lambda + \log P_2(b|a)) + \exp((1-\lambda) + \log P_1(b))\right) \\
&+ \log\left(\exp(\lambda + \log P_2(a|b)) + \exp((1-\lambda) + \log P_1(a))\right)
\end{aligned}$$

- The score of the input will the **sum** of probabilities across all paths that successfully process the input.
- What happens if we use the tropical semiring?

10.5 Finite state transducers

FSAs and WFSAs apply to single strings. FSTs and WFSTs apply to pairs of string.

	acceptor	transducer
unweighted	FSA: $\Sigma^* \rightarrow \{0, 1\}$	WFSA: $\Sigma^* \rightarrow \mathbb{R}$
weighted	FST: $\Sigma^* \rightarrow \Sigma^*$	WFST: $\Sigma^* \rightarrow \langle \Sigma^*, \mathbb{R} \rangle$

FSTs define **regular relations** over pairs of strings. We can think of them in a few different ways:

- **Recognizer**: accepts string pairs iff they are in the relation
- **Translator**: reads an input, produces an output

Formally, a finite-state transducer $M = \langle Q, \Sigma, \Delta, q_0, F, \delta, \sigma \rangle$ consists of:

- A finite set of states $Q = \{q_0, q_1, \ldots, q_n\}$
- Finite alphabets Σ for input symbols and Δ for output symbols
- Initial state $q_0 \in Q$ and final states $F \subseteq Q$
- A state transition function $\delta : \langle Q \times \Sigma^* \rangle \rightarrow 2^Q$
- A string transition function $\sigma : \langle Q \times \Sigma^* \rangle \rightarrow 2^{\Delta^*}$

Unlike NFSAs, not all 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)).

- A zeroth-order translation system could be made from a single state and a set of self-transitions: $Q_0 \xrightarrow[\textit{el}]{\textit{the}} Q_0$, $Q_0 \xrightarrow[\textit{los}]{\textit{the}} Q_0$, $Q_0 \xrightarrow[\textit{libro}]{\textit{book}} Q_0$, $Q_0 \xrightarrow[\textit{libros}]{\textit{books}} Q_0$,...
- First-order translation would require a state per word in the "input" vocabulary: $Q_0 \xrightarrow[\epsilon]{\textit{the}} Q_{\text{the}} \xrightarrow[\textit{los libros}]{\textit{books}} Q_0$, $Q_{\text{the}} \xrightarrow[\textit{el libro}]{\textit{book}} Q_0$.
- Inflectional morphology and orthography:

$$Q_0 \xrightarrow[\textit{wit}]{\textit{wit}} Q_{\text{regular}} \xrightarrow[\textit{+s}]{\text{+PL}} Q_1$$
$$Q_0 \xrightarrow[\textit{wish}]{\textit{wish}} Q_{\text{needs-e}} \xrightarrow[\textit{+es}]{\text{+PL}} Q_1$$

10.6 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.
- For example, we could augment the translation transducers defined above to allow alternative possible translations for a single word.
- The same semiring operations in WFSAs apply here too.

	acceptor	transducer
unweighted	FSA: $\Sigma^* \to \{0,1\}$	WFSA: $\Sigma^* \to \mathbb{R}$
weighted	FST: $\Sigma^* \to \Sigma^*$	WFST: $\Sigma^* \to \langle \Sigma^*, \mathbb{R} \rangle$

Example : General edit distance computer.

- $Q_0 \xrightarrow[a]{a} Q_0 : 0$
- $Q_0 \xrightarrow[\epsilon]{a} Q_0 : 1$
- $Q_0 \xrightarrow[a]{\epsilon} Q_0 : 1$

Operations on FSTs

- Closed under **union**
- Closed under **inversion**, which switches input and output labels.
- Closed under **projection**, because FSAs are a special case of FSTs
- Not closed under **difference**, **complementation**, and **intersection**.
- Closed under **composition**.

FST composition is the basis for implementing the noisy channel model in FSTs, and can be used to support dozens of cool applications.

Finite state composition

Suppose we have a transducer T_1 from language I_1 to O_1, and another transducer T_2 from O_1 to O_2. The composition $T_1 \circ T_2$ is an FST from I_1 to O_2.

- 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:

$$(T_1 \circ T_2)(x, y) = \bigoplus_{z \in \Sigma^*} T_1(x, z) \otimes T_2(z, y) \tag{10.3}$$

- Designing algorithms for composition is relatively straightforward if there are no epsilon transitions; otherwise it's more challenging Allauzen et al. (2009).

The simplest example

- $T_1 : Q_0 \overset{x}{\underset{a}{\rightarrow}} Q_0, Q_0 \overset{y}{\underset{b}{\rightarrow}} Q_0$
- $T_2 : Q_1 \xrightarrow{a} Q_1, Q_1 \xrightarrow{b} Q_2, Q_2 \xrightarrow{b} Q_2$
- $T_1 \circ T_2 : Q_1 \xrightarrow{x} Q_1, Q_1 \xrightarrow{y} Q_2, Q_2 \xrightarrow{y} Q_2$

For simplicity T_2 is written as a finite-state acceptor, not a transducer. Acceptors are a special case of transducers.

If we had weights, they would be combined through $\otimes$.

10.7 Applications of composition

Stemming

As discussed on Tuesday, information retrieval systems that only return exact matches aren't very useful.

- Suppose you query: *is it medically safe to kiss my cat on the lips*
- You want to get a hit even for documents like: *on the medical safety of kissing cats*.

- In morphologically complex languages like Hebrew, these differences could cause early IR systems to miss more than 90% of relevant documents Choueka (1989)!

- Stemming improves the **recall** of information retrieval systems by converting all tokens of *kissing* to *kiss*, etc.

The **Porter Stemmer** is a very popular stemming program. It is written as a set of rules, which are applied in stages, e.g.

- if the word ends in *-ing* and the preceeding part contains a vowel, delete the ending. (*hopping* $\rightarrow$ *hopp*)

- if the remaining part contains double letters (besides ss, ll, or zz), remove one (*hopp* $\rightarrow$ *hop*)

We can think of these rules as a sequence of deterministic finite state transducers:

- We can build a transducer to strip off endings like *-ing*, using two states to indicate whether we have yet seen a vowel

- Next we can build a transducer to strip off double letter endings, with exceptions for s, l, and z.

- We can **compose** these transducers into a single machine.

Back to edit distance

Remember the general-purpose edit distance FST? What happens if we compose that with a chain FSA for a word? We get the word-specific edit distance computer we talked about earlier.

- Composing an FST with a FSA yields a FSA.

- In many applications, we can 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.

Machine translation

- Simple models of machine translation can be implemented as finite-state transducers.

- Recall the example: *the books* / *los libros*. Here we are interested in modeling English-to-Spanish translation, $P(S|E)$.

  – Earlier we proposed a multi-state translation model to deal with the impact of pluralization on the Spanish article *los*

  – If we build our translator by composing a Spanish language model $P(S)$ with an Spanish-to-English transducer $P(E|S)$, this is not necessary. We can use the **noisy channel model**.

  – The $P(E|S)$ model can be a single state, as long as $P(S)$ models bigrams. A bigram model will tell us that $P(\textit{el libros}) \ll P(\textit{los libros})$.

  $$P(\textit{los libros}, \textit{the books}) = P_S(\textit{los}|\star) \otimes P_{E|S}(\textit{the}|\textit{los}) \otimes P_S(\textit{libros}|\textit{los}) \otimes P_{E|S}(\textit{books}|\textit{libros})$$

  – The composed FST can thus overcome its simplistic model of translation by having a more intelligent language model. This is useful, because language models can be trained without labeled data, while translation models cannot.

  – The general translation model looks like this: $T_S \circ T_{E|S}$. If we right-compose with an FSA representing the English string, we obtain a WFSA that scores all Spanish strings as $P(E, S) \propto P(S|E)$.

  – The reading Knight and May (2009) describes how we can add capabilities like adding and re-ordering words.

Morphological analysis

Recall that when we talked about morphology, there were several types of interacting rules:

1. To pluralize regular words, add an *s*; but if the word ends in *sh*, you have to add *es*.

2. To conjugate 3rd-person singular, add an *s*; but if the word ends in *sh*, you have to add *es*.

Pluralization and conjugation are different morphological systems, but once they have decided to append an s, the subsequent orthographical rules are the same.

This suggests an intermediate representation:

- *dish*+PL $\rightarrow$ *dish∧s#* $\rightarrow$ *dishes*
- *fish*+PL $\rightarrow$ *fish#* $\rightarrow$ *fish*
- *fish*+3S $\rightarrow$ *fish∧s#* $\rightarrow$ *fishes*
- Special symbols for morpheme and word boundaries
- The conjugation and pluralization FSTs only need to know what affix to add, and where to add it.
- Then an orthography FST takes over, and figures out how the morphemes should be combined.
- Finite-state composition allows us to automatically build a single machine for conjugation and pluralization, incorporating both the selection of affixes and orthographic constraints.
- Note that we have only described how to *generate* the English text. But if we compose this machine with a chain FSA representing an observed string, then we obtain an FSA where the set of accepted strings reveal the acceptable morphological analyses.
- For example, an utterance like *wishes* could have been produced by *wish/N*+3S or by *wish/V*+PL; both will be accepted by the resulting FSA.
- Suppose we want to distinguish the most likely morphological analysis given the context.
  - First, we add a loop back to the beginning in the morphological FST, so that we can accept multiple words.
  - Now we can build something like a language model WFST, but here we need probabilities of morphological analyses rather than just words. We might want to decompose this like

$$P(stem, affixes|context) \approx P(stem|context)P(affix|context) \quad (10.4)$$

  - If we have a morphologically annotated corpus, we can make smoothed relative frequency estimates of these probabilities. If we don't, what do we do?
  - The morphological analyses are missing data, so we might be able to do EM:
    * **E-step**: Decode all observed string sequences in the corpus, given the current probabilities.
    * **M-step**: Treat decoding as ground truth, update probabilities
    * Note: this is "Viterbi" EM, (a.k.a. "Hard" EM), because we are not computing probabilities over all possible decodings. We could do that using something called the expectation semiring Eisner (2002).

Context-sensitive spelling correction

Swype text entry in my old android phone does not consider context, much to my annoyance.

- I mean: *Prepare lecture for my class*
- It says: *Prepare lecture foie my class*

That's not smart. The bigram probabilities $P(\textit{lecture foie})$ and $P(\textit{foie my})$ should be ridiculously low (even dumber, it doesn't know that the unigram probability $P(\textit{foie}) \ll P(\textit{for})$, but that's another story...).

Rather than composing our edit distance computer with a single chain FSA, we can compose it with a language model FST — just like the language model FSA described above, but with output identical to the input.

The resulting machine is an FST with one state per word type.

- The cost of the transition from Q_a to Q_b on input s is the semiring product $\otimes$ of two costs:
  - The transition cost from a to b in the language model
  - The "emission" cost from s to a in the edit distance machine. Depending on exactly how we're getting our input, we could memorize these costs (for all pairs of words) and work at the token level, rather than character level. But there's no conceptual difference; alternatively you could think about character-level edit distance FSTs sitting at each state.

- Given an input sequence s, we compose

  select the spelling-corrected string t with the greatest value,

$$\hat{t} = \max_t \bigoplus_{\pi} s \rightsquigarrow_{\pi} t, \tag{10.5}$$

  where π is a path over input s and output t.

  – Why are there multiple paths from s to t? In the edit distance machine, we can always model matching input/output symbols as an insertion and a deletion.

  – Whether we care about these alternative paths depends on the semiring. In the tropical semiring, the solution is simply the minimum-cost path.

Norvig's spelling corrector

We haven't said much about where the weights come from. The bigram language model is probabilistic, and we can compute $\delta(q_s, q_t, t, t)$ as $-\log P(t|s)$. What about the edit distance model?

Before we get into that, here's an alternative approach, from Peter Norvig `http://norvig.com/spell-correct.html` (highly recommended). It may help explain how google is able to quickly and accurately spell-check your queries.

The approach can't easily be framed in exactly terms of the FST/semiring formalism, but it's close:

- Check if the word itself is in the dictionary. If so, return it.
- Else, check if any edit-distance=1 corrections is in the dictionary. If so, return the one with the highest unigram count.
- Else try corrections with edit-distance = 2.

The essay discusses extensions to bigrams a probabilistic model of errors. One way to build such a model is to look at data.

- Norvig suggests the Birbeck spelling error corpus, `http://norvig.com/spell-correct.html`.

  – Such a resource would tell us the likelihood of a word s being mispelled as t.

- But new mispellings are always possible (even inevitable)

- An alternative would be to parametrize the edit distance model with more specific probabilities: P(insertion), P(deletion), etc.
  - Maximum-likelihood estimation would compute

$$P(\text{insertion}) = \frac{\text{count}(\text{insertion})}{\text{count}(\text{all-characters})} \tag{10.6}$$

  - But we will probably never get a corpus that gives us these counts.
  - Can we bootstrap our way to success? Suppose we could just guess the probabilities.
    * We could find the best path for each example in the training set, and keep track of the counts of insertions and deletions in these paths.

$$\begin{aligned}\hat{\pi}_{s,t} =&\arg\min_{\pi} \text{cost}(s \rightsquigarrow_{\pi} t)\\ \text{count}(\text{insertion}) =& \sum_{\langle s,t\rangle \in \mathcal{D}} \text{count}(\text{insertion}, \hat{\pi}_{s,t})\end{aligned}$$

    * More probabilistically, we could compute the likelihood of each path, and use these likelihood to find the *expected* counts:

$$\text{count}(\text{insertion}) = \sum_{\langle s,t\rangle \in \mathcal{D}} \sum_{\pi} \frac{P(s \rightsquigarrow_{\pi} t)}{\sum_{\pi'} P(s \rightsquigarrow_{\pi'} t)} \text{count}(\text{insertion}, \pi_{s,t})$$

    * Once we have the counts, we can go back and re-estimate the insertion probability (and all the other probabilities).

- This type of bootstrapping is another example of **expectation-maximization**.
  - We introduced EM in the simpler case of clustering.
    * Each document had a distribution over clusters $q(z)$
    * Each cluster had parameters θ.
    * The E-step computed $q(z)$, the M-step optimized θ.
  - Here, the Q distribution is over paths $Q(\pi)$.
  - In the E-step, we can compute $Q(\pi)$ given estimates of the parameters $P(\text{insertion})$.

  - In the M-step, we can update the parameters $P(\text{insertion})$ from expected counts under $Q(\pi)$.

- The number of paths can be very large, often too large to store.

  - In **Viterbi EM**, we just use the best path. This can work very well.
  - Alternatively, we can use the **expectation semiring** so that the "score" of the weighted path sum includes the expected counts Eisner (2001).

    Each edge is a tuple of a probability and a vector of expected counts, $\langle p_i, p_i v_i \rangle$

$$
\begin{aligned}
\langle p_i, p_i v_i \rangle \otimes \langle p_j, p_j v_j \rangle &= \langle p_i p_j, p_j p_i v_i + p_i p_j v_j \rangle = \langle p_k, p_k v_k \rangle \\
\langle p_i, v_i \rangle \oplus \langle p_j, v_j \rangle &= \langle p_i + p_j, v_i + v_j \rangle
\end{aligned}
$$

  - If we can compute the semiring shortest path ($\mathcal{O}(n^3)$ at worst), we can compute the expected counts!

Speech Recognition

Speech recognition is yet another application of finite-state methods in NLP. It takes the idea of intermediate representations even further.

We want a mapping from intended words to observed acoustics. This can be seen as a multilevel process:

- G: WFST which generates English sentences: *I went to the bank*

- L: WFST which converts each word to context-indendent phonemes (WFST. Pronunciation dictionaries have this information): *Ay w eh n t t uw ...*

- C: WFST which converts context-independent phonemes to context-dependent phonemes: *Ay w eh n t uw ...*

- A: WFST which converts context-dependent phonemes to acoustic observations

By composing the FST $G \circ L \circ C \circ A$ with an FSA representing the observed acoustics O, we obtain a single WFSA which scores proposed English sentences for the observations O.

Inference The composed FSA would be ridiculously huge. Beam pruning is a technique for pruning away paths which are extremely unlikely, resulting in a faster, smaller FST.

Estimation It's really hard to get labeled data for all of these levels, for example context-dependent phones. We can treat this as a hidden variable and estimate it using EM.

Software There are mature software toolkits for working with finite state machines. OpenFST is a C++ package which I have had some experience with; it's fast and relatively well-documented. XFST and Carmel are other options.

Bibliography

Allauzen, C., Riley, M., and Schalkwyk, J. (2009). A generalized composition algorithm for weighted finite-state transducers. In *Proceedings of Interspeech*.

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.

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.

Bottou, L. (1998). Online learning and stochastic approximations. *On-line learning in neural networks*, 17:9.

Boyd, S. and Vandenberghe, L. (2004). *Convex Optimization*. Cambridge University Press, New York, NY, USA.

Burges, C. J. (1998). A tutorial on support vector machines for pattern recognition. *Data mining and knowledge discovery*, 2(2):121–167.

Choueka, Y. (1989). Responsa: A full-text retrieval system with linguistic processing for a 65-million word corpus of jewish heritage in hebrew. *Data Engineering*, 12(4):22–31.

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.

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. (2013). Notes on natural language processing. `http://www.cs.columbia.edu/~mcollins/notes-spring2013.html`.

Collins, M. and Duffy, N. (2001). Convolution kernels for natural language. In *Advances in neural information processing systems*, pages 625–632.

Collobert, R. and Weston, J. (2008). A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167. ACM.

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. (2003). Ultraconservative online algorithms for multiclass problems. *The Journal of Machine Learning Research*, 3:951–991.

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.

Dyer, C. (2014). Notes on adagrad. `www.ark.cs.cmu.edu/cdyer/adagrad.pdf`.

Eisner, J. (2001). Expectation semirings: Flexible em for learning finite-state transducers. In *Proceedings of the ESSLLI workshop on finite-state methods in NLP*.

Eisner, J. (2002). Parameter estimation for probabilistic finite-state transducers. In *Proceedings of the Association for Computational Linguistics (ACL)*, pages 1–8.

Figueiredo, M., Graça, J., Martins, A., Almeida, M., and Coelho, L. P. (2013). LXMLS lab guide. `http://lxmls.it.pt/2013/guide.pdf`.

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.

Hastie, T., Tibshirani, R., Friedman, J., Hastie, T., Friedman, J., and Tibshirani, R. (2009). *The elements of statistical learning*, volume 2. Springer.

Jaeger, T. F. (2010). Redundancy and reduction: Speakers manage syntactic information density. *Cognitive psychology*, 61(1):23–62.

Jurafsky, D. and Martin, J. H. (2009). *Speech and Language Processing (2nd Edition) (Prentice Hall Series in Artificial Intelligence)*. Prentice Hall, 2 edition.

Karttunen, L. and Beesley, K. R. (2001). A short history of two-level morphology. *ESSLLI-2001 Special Event titled Twenty Years of Finite-State Morphology*.

Karttunen, L. and Beesley, K. R. (2005). Twenty-five years of finite-state morphology. *Inquiries Into Words, a Festschrift for Kimmo Koskenniemi on his 60th Birthday*, pages 71–83.

Knight, K. and May, J. (2009). Applications of weighted automata in natural language processing. In *Handbook of Weighted Automata*, pages 571–596. Springer.

Manning, C. D. and Schütze, H. (1999). *Foundations of statistical natural language processing*. MIT press.

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.

Mnih, A. and Hinton, G. E. (2008). A scalable hierarchical distributed language model. In *Neural Information Processing Systems (NIPS)*, pages 1081–1088.

Mohri, M., Pereira, F., and Riley, M. (2002). Weighted finite-state transducers in speech recognition. *Computer Speech & Language*, 16(1):69–88.

Murphy, K. P. (2012). *Machine Learning: A Probabilistic Perspective*. The MIT Press.

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.*, 19.

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.

Pereira, F. (2000). Formal grammar and information theory: together again? *Philosophical Transactions of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, 358(1769):1239–1253.

Roark, B., Saraclar, M., and Collins, M. (2007). Discriminative¡ i¿ n¡/i¿-gram language modeling. *Computer Speech & Language*, 21(2):373–392.

Rosenfeld, R. (1996). A maximum entropy approach to adaptive statistical language modelling. *Computer Speech & Language*, 10(3):187–228.

Saul, L. and Pereira, F. (1997). Aggregate and mixed-order markov models for statistical language processing. In *emnlp*.

Schmid, H. (1994). Probabilistic part-of-speech tagging using decision trees. In *Proceedings of the international conference on new methods in language processing*, volume 12, pages 44–49. Manchester, UK.

Shwartz, S. S., Singer, Y., and Srebro, N. (2007). Pegasos: Primal estimated sub-GrAdient SOlver for SVM. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 807–814.

Sra, S., Nowozin, S., and Wright, S. J. (2012). *Optimization for machine learning*. MIT Press.

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.

Zelenko, D., Aone, C., and Richardella, A. (2003). Kernel methods for relation extraction. *The Journal of Machine Learning Research*, 3:1083–1106.

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.