# Linear Models for Statistical Natural Language Processing

Jacob Eisenstein

August 28, 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 $K = \#\lvert\mathcal{Y}\rvert$ |
| $\boldsymbol{f}(\boldsymbol{x}_i, y_i)$ | 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: $\boldsymbol{x}_i = [0\ 1\ 1\ 0\ 0\ 2\ 0\ 1\ 13\ 0 \ldots]^\mathsf{T}$, 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:

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

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

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

$$\ldots \tag{3.4}$$

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

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,

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

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.

$$\begin{aligned} P(A_i \cup A_j) &= P(A_i) + P(A_j - (A_i \cap A_j)) && (3.7)\\ P(A_j) &= P(A_j - (A_i \cap A_j)) + P(A_i \cap A_j) && (3.8)\\ P(A_j - (A_i \cap A_j)) &= P(A_j) - P(A_i \cap A_j) && (3.9)\\ P(A_i \cup A_j) &= P(A_i) + P(A_j) - P(A_i \cap A_j) && (3.10) \end{aligned}$$

- 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 $E[g(x)] = \sum_{x \in \mathcal{X}} g(x)\text{p}(x)$.

Expectations are easiest to think about in terms of probability distributions over discrete events:

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

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

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

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, $\text{p}(\boldsymbol{x}_{1:N}, \boldsymbol{y}_{1:N})$, where $\langle \boldsymbol{x}_i, y_i \rangle$ is a labeled instance.

We first need to define the probability $\text{p}(\boldsymbol{x}, y)$. 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 $y_i \sim \text{Categorical}(\mu)$
  - draw the vector of counts $\boldsymbol{x}_i \sim \text{Multinomial}(\phi_{y_i})$,

[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 $\mu$ and $\phi$.

When we write $y_i \sim \text{Categorical}(\mu)$, that means $y_i$ is a stochastic draw from a categorical distribution with **parameter** $\mu$. A categorical distribution is just like a weighted die: $\mathrm{p}_{\text{cat}}(y; \mu) = \mu_y$, where $\mu_y$ 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 $\phi$, 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 $\phi$, 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 $\mu$ and $\boldsymbol{\phi}$ (defined later).

## Prediction

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

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

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.

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

where

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

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 $\boldsymbol{\theta}^\mathsf{T} \boldsymbol{f}(\boldsymbol{x}, y)$ 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 $\phi$, 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 $\alpha$ 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 $\phi$ as a random variable (rather than as a parameter). The resulting estimate is called *maximum a posteriori*, or MAP.

Smoothing reduces **variance**, but it takes us away from the maximum-likelihood estimate: it imposes a **bias** (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 $\alpha \rightarrow \infty$, we have zero variance: it is the same classifier no matter what data we see! But the bias of such a classifier will be high.
- Navigating this tradeoff is hard. But in general, as you have more data, variance is less of a problem, so you just go for low bias.

## 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 $\alpha$ in Naive Bayes
- the regularization $\lambda$ 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 $\phi_y$). Formally, we assume conditional independence:

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

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

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

  or...

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

  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 $\text{p}(\boldsymbol{x}, \boldsymbol{y})$ via a *generative model*
  - Prediction: $\hat{y} = \arg\max_y \text{p}(\boldsymbol{x}_i, y)$
  - Learning:

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

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

- Bias-variance tradeoff: MLE is high-variance, so add smoothing pseudo counts $\alpha$. 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.

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

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

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:

$$\overline{\boldsymbol{\theta}} = \frac{1}{T} \sum_{t}^{T} \boldsymbol{\theta}_t$$
$$y = \arg\max_{y} \overline{\boldsymbol{\theta}}^{\mathsf{T}} \boldsymbol{f}(\boldsymbol{x}, y)$$

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:

$$\log \mathrm{p}(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{\theta}) = \sum_{i} \log \mathrm{p}(\boldsymbol{x}_i, y_i; \boldsymbol{\theta}) \tag{5.4}$$
$$\ell_{\text{NB}}(\boldsymbol{\theta}; \boldsymbol{x}_i, y_i) = -\log \mathrm{p}(\boldsymbol{x}_i, y_i; \boldsymbol{\theta}) \tag{5.5}$$
$$\hat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \sum_{i} \ell_{\text{NB}}(\boldsymbol{\theta}, \boldsymbol{x}_i, y_i) \tag{5.6}$$

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.

- $\ell_{NB}$ 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.

## 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

[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)

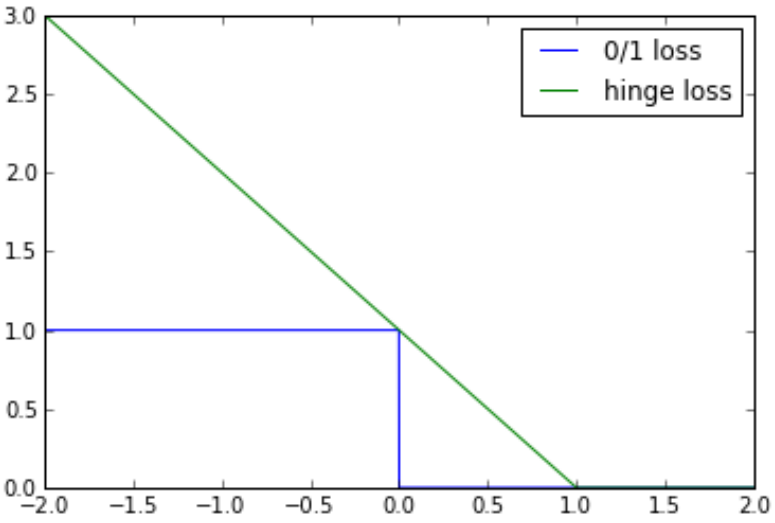


Figure 5.1: Hinge and perceptron loss functions

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,

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

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

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

[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.

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 $\xi_i$. 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 $C$.

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

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 $C \rightarrow \infty$, 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.
- **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

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

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,

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

$$\log \mathrm{p}(y|\boldsymbol{x}) = \sum_i \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}_i, y_i) - \log \sum_{y'} \exp \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}_i, y') \tag{5.17}$$

$$\hat{\boldsymbol{\theta}} = \arg\max_{\boldsymbol{\theta}} \sum_i \log \mathrm{p}(y_i|\boldsymbol{x}_i; \boldsymbol{\theta}) \tag{5.18}$$

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]

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

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

$$\ell = \sum_i \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}_i, y_i) - \log \sum_{y'} \exp \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}_i, y') \tag{5.21}$$

$$\frac{\partial \ell}{\partial \boldsymbol{\theta}} = \sum_i \boldsymbol{f}(\boldsymbol{x}_i, y_i) - \frac{\sum_{y'} \exp \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}_i, y') \boldsymbol{f}(\boldsymbol{x}_i, y')}{\sum_{y''} \exp \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}_i, y')} \tag{5.22}$$

$$= \sum_i \boldsymbol{f}(\boldsymbol{x}_i, y_i) - \sum_{y'} \frac{\exp \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}_i, y')}{\sum_{y''} \exp \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}_i, y')} \boldsymbol{f}(\boldsymbol{x}_i, y') \tag{5.23}$$

$$= \sum_i \boldsymbol{f}(\boldsymbol{x}_i, y_i) - \sum_{y}{}' \mathrm{p}(y'|\boldsymbol{x}_i; \boldsymbol{\theta}) \boldsymbol{f}(\boldsymbol{x}_i, y') \tag{5.24}$$

$$= \sum_i \boldsymbol{f}(\boldsymbol{x}_i, y_i) - E[\boldsymbol{f}(\boldsymbol{x}_i, y')] \tag{5.25}$$

[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.

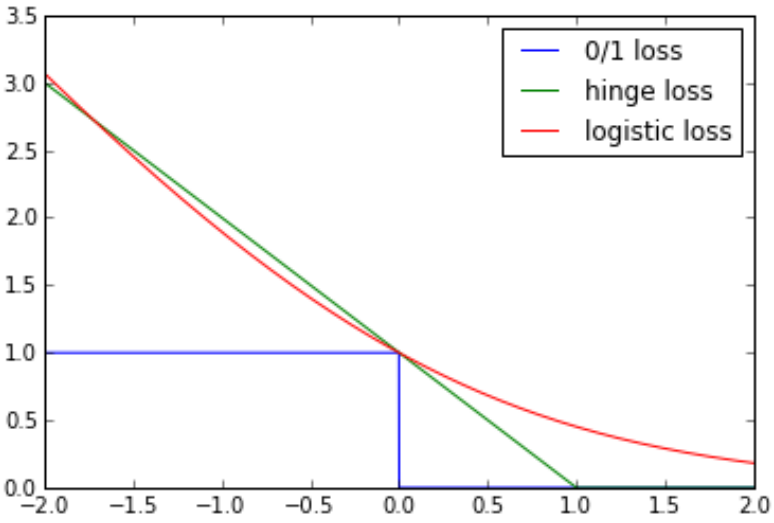


Figure 5.2: Hinge, perceptron, and logistic loss functions

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

$$\ell = \sum_i \boldsymbol{\theta}^\mathsf{T} \boldsymbol{f}(\boldsymbol{x}_i, y_i) - \log \sum_{y'} \exp \boldsymbol{\theta}^\mathsf{T} \boldsymbol{f}(\boldsymbol{x}_i, y') + \frac{\lambda}{2}||\boldsymbol{\theta}||_2^2$$

$$\frac{\partial \ell}{\partial \boldsymbol{\theta}} = \sum_i \boldsymbol{f}(\boldsymbol{x}_i, y_i) - E[\boldsymbol{f}(\boldsymbol{x}_i, y')] - \lambda\boldsymbol{\theta}.$$

## Optimization

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

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

- 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

$$H_{i,j} = \frac{\partial^2}{\partial w_i \partial w_j} \ell, \tag{5.26}$$

  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 stochastic online approximation to the overall gradient:

$$\begin{aligned}
\boldsymbol{\theta}^{(t+1)} \leftarrow & \boldsymbol{\theta}^{(t)} - \eta_t \nabla_{\boldsymbol{\theta}} \ell(\boldsymbol{\theta}^{(t)}, \boldsymbol{x}, \boldsymbol{y}) \\
= & \boldsymbol{\theta}^{(t)} - \eta_t \frac{1}{N} (\lambda \boldsymbol{\theta}^{(t)} - \sum_i^N \boldsymbol{f}(\boldsymbol{x}_i, y_i) - E[\boldsymbol{f}(\boldsymbol{x}_i, y')]) \\
= & (1 - \lambda \eta_t) \boldsymbol{\theta}^{(t)} + \eta_t \frac{1}{N} \sum_i^N \boldsymbol{f}(\boldsymbol{x}_i, y_i) - E[\boldsymbol{f}(\boldsymbol{x}_i, y')]
\end{aligned}$$

where $\eta_t$ is the **stepsize** at time $t$.

- If we set $\eta_t = \eta_0 t^{-\alpha}$ for $\alpha \in [1, 2]$, we have guaranteed convergence.
- We can also just fix $\eta_t$ to a small value. In either case, we have to tune this parameter on a development set.
- Note how similar this update is to the perceptron.

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

**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:

$$\boldsymbol{g}_t = -\boldsymbol{f}(\boldsymbol{x}_i, y_i) + \sum_{y'} \mathrm{p}(y' \mid \boldsymbol{x}_i)\boldsymbol{f}(\boldsymbol{x}_i, y_i) + \lambda\boldsymbol{\theta} \tag{5.27}$$

$$\theta_j^{(t+1)} \leftarrow \theta_j^{(t)} - \frac{\eta}{\sqrt{\sum_{t'}^{t} g_{t,j}^2}} g_{t,j}, \tag{5.28}$$

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 $\eta$, 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 $\boldsymbol{\theta}^\mathsf{T}\boldsymbol{x}$ 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:

$$\forall j, \sum_i f_j(\boldsymbol{x}_i, y_i) = \sum_i \sum_y \mathrm{p}(y \mid \boldsymbol{x}_i; \boldsymbol{\theta}) f_j(\boldsymbol{x}_i, y) \tag{5.29}$$

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

$$H(P) = -\sum_x \tilde{\mathrm{p}}(x) \sum_y \mathrm{p}(y|x) \log \mathrm{p}(y|x), \tag{5.30}$$

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:

$$\begin{aligned} \max_{\boldsymbol{\theta}} \quad & -\sum_{\boldsymbol{x}} \tilde{\mathrm{p}}(\boldsymbol{x}) \sum_y \mathrm{p}(y|\boldsymbol{x}; \boldsymbol{\theta}) \log \mathrm{p}(y|\boldsymbol{x}; \boldsymbol{\theta}) \\ s.t. \quad & \forall j, \sum_i f_j(\boldsymbol{x}_i, y_i) = \sum_i \sum_y \mathrm{p}(y|\boldsymbol{x}_i; \boldsymbol{\theta}) f_j(\boldsymbol{x}_i, y) \end{aligned}$$

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).

- 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.

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

- 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 $\alpha$ | 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

# Bibliography


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.

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.

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.

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.

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

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

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.

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.

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