Chapter 2

Linear text classification

We begin with the problem of **text classification**: given a text document, assign it a discrete label $y \in \mathcal{Y}$, where $\mathcal{Y}$ is the set of possible labels. Text classification has many applications, from spam filtering to the analysis of electronic health records. This chapter describes some of the most well known and effective algorithms for text classification, from a mathematical perspective that should help you understand what they do and why they work. Text classification is also a building block in more elaborate natural language processing tasks. For readers without a background in machine learning or statistics, the material in this chapter will take more time to digest than most of the subsequent chapters. But this investment will pay off as the mathematical principles behind these basic classification algorithms reappear in other contexts throughout the book.

2.1 The bag of words

To perform text classification, the first question is how to represent each document, or instance. A common approach is to use a column vector of word counts, e.g., $\boldsymbol{x} = [0, 1, 1, 0, 0, 2, 0, 1, 13, 0 \ldots]^\top$, where x_j is the count of word j. The length of $\boldsymbol{x}$ is $V \triangleq |\mathcal{V}|$, where $\mathcal{V}$ is the set of possible words in the vocabulary. In linear classification, the classification decision is based on a weighted sum of individual feature counts, such as word counts.

The object $\boldsymbol{x}$ is a vector, but it is often called a **bag of words**, because it includes only information about the count of each word, and not the order in which the words appear. With the bag of words representation, we are ignoring grammar, sentence boundaries, paragraphs — everything but the words. Yet the bag of words model is surprisingly effective for text classification. If you see the word *whale* in a document, is it fiction or nonfiction? What if you see the word *molybdenum*? For many labeling problems, individual words can be strong predictors.

To predict a label from a bag-of-words, we can assign a score to each word in the vocabulary, measuring the compatibility with the label. For example, for the label FICTION, we might assign a positive score to the word *whale*, and a negative score to the word *molybdenum*. These scores are called **weights**, and they are arranged in a column vector $\boldsymbol{\theta}$.

Suppose that you want a multiclass classifier, where $K \triangleq |\mathcal{Y}| > 2$. For example, you might want to classify news stories about sports, celebrities, music, and business. The goal is to predict a label $\hat{y}$, given the bag of words $\boldsymbol{x}$, using the weights $\boldsymbol{\theta}$. For each label $y \in \mathcal{Y}$, we compute a score $\Psi(\boldsymbol{x}, y)$, which is a scalar measure of the compatibility between the bag-of-words $\boldsymbol{x}$ and the label y. In a linear bag-of-words classifier, this score is the vector inner product between the weights $\boldsymbol{\theta}$ and the output of a **feature function** $\boldsymbol{f}(\boldsymbol{x}, y)$,

$$\Psi(\boldsymbol{x}, y) = \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y) = \sum_j \theta_j f_j(\boldsymbol{x}, y). \quad [2.1]$$

As the notation suggests, $\boldsymbol{f}$ is a function of two arguments, the word counts $\boldsymbol{x}$ and the label y, and it returns a vector output. For example, given arguments $\boldsymbol{x}$ and y, element j of this feature vector might be,

$$f_j(\boldsymbol{x}, y) = \begin{cases} x_{\textit{whale}}, & \text{if } y = \text{FICTION} \\ 0, & \text{otherwise} \end{cases} \quad [2.2]$$

This function returns the count of the word *whale* if the label is FICTION, and it returns zero otherwise. The index j depends on the position of *whale* in the vocabulary, and of FICTION in the set of possible labels. The corresponding weight θ_j then scores the compatibility of the word *whale* with the label FICTION.[1] A positive score means that this word makes the label more likely.

The output of the feature function can be formalized as a vector:

$$\boldsymbol{f}(\boldsymbol{x}, y = 1) = [\boldsymbol{x}; \underbrace{0; 0; \ldots; 0}_{(K-1)\times V}] \quad [2.3]$$

$$\boldsymbol{f}(\boldsymbol{x}, y = 2) = [\underbrace{0; 0; \ldots; 0}_{V}; \boldsymbol{x}; \underbrace{0; 0; \ldots; 0}_{(K-2)\times V}] \quad [2.4]$$

$$\boldsymbol{f}(\boldsymbol{x}, y = K) = [\underbrace{0; 0; \ldots; 0}_{(K-1)\times V}; \boldsymbol{x}], \quad [2.5]$$

where $[\underbrace{0; 0; \ldots; 0}_{(K-1)\times V}]$ is a column vector of $(K-1) \times V$ zeros, and the semicolon indicates vertical concatenation. For each of the K possible labels, the feature function returns a

[1] In practice, both $\boldsymbol{f}$ and $\boldsymbol{\theta}$ may be implemented as a dictionary rather than vectors, so that it is not necessary to explicitly identify j. In such an implementation, the tuple (*whale*, FICTION) acts as a key in both dictionaries; the values in $\boldsymbol{f}$ are feature counts, and the values in $\boldsymbol{\theta}$ are weights.

vector that is mostly zeros, with a column vector of word counts $\boldsymbol{x}$ inserted in a location that depends on the specific label y. This arrangement is shown in Figure 2.1. The notation may seem awkward at first, but it generalizes to an impressive range of learning settings, particularly **structure prediction**, which is the focus of Chapters 7-11.

Given a vector of weights, $\boldsymbol{\theta} \in \mathbb{R}^{VK}$, we can now compute the score $\Psi(\boldsymbol{x}, y)$ by Equation 2.1. This inner product gives a scalar measure of the compatibility of the observation $\boldsymbol{x}$ with label y.[2] For any document $\boldsymbol{x}$, we predict the label $\hat{y}$,

$$\hat{y} = \operatorname*{argmax}_{y \in \mathcal{Y}} \Psi(\boldsymbol{x}, y) \quad [2.6]$$

$$\Psi(\boldsymbol{x}, y) = \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y). \quad [2.7]$$

This inner product notation gives a clean separation between the *data* ($\boldsymbol{x}$ and y) and the *parameters* ($\boldsymbol{\theta}$).

While vector notation is used for presentation and analysis, in code the weights and feature vector can be implemented as dictionaries. The inner product can then be computed as a loop. In python:

```
def compute_score(x,y,weights):
    total = 0
    for feature,count in feature_function(x,y).items():
        total += weights[feature] * count
    return total
```

This representation is advantageous because it avoids storing and iterating over the many features whose counts are zero.

It is common to add an **offset feature** at the end of the vector of word counts $\boldsymbol{x}$, which is always 1. We then have to also add an extra zero to each of the zero vectors, to make the vector lengths match. This gives the entire feature vector $\boldsymbol{f}(\boldsymbol{x}, y)$ a length of $(V + 1) \times K$. The weight associated with this offset feature can be thought of as a bias for or against each label. For example, if we expect most emails to be spam, then the weight for the offset feature for $y = \text{SPAM}$ should be larger than the weight for the offset feature for $y = \text{NOT-SPAM}$.

Returning to the weights $\boldsymbol{\theta}$, where do they come from? One possibility is to set them by hand. If we wanted to distinguish, say, English from Spanish, we can use English and Spanish dictionaries, and set the weight to one for each word that appears in the

[2] Only $V \times (K - 1)$ features and weights are necessary. By stipulating that $\Psi(\boldsymbol{x}, y = K) = 0$ regardless of $\boldsymbol{x}$, it is possible to implement any classification rule that can be achieved with $V \times K$ features and weights. This is the approach taken in binary classification rules like $y = \text{Sign}(\boldsymbol{\beta} \cdot \boldsymbol{x} + a)$, where $\boldsymbol{\beta}$ is a vector of weights, a is an offset, and the label set is $\mathcal{Y} = \{-1, 1\}$. However, for multiclass classification, it is more concise to write $\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y)$ for all $y \in \mathcal{Y}$.

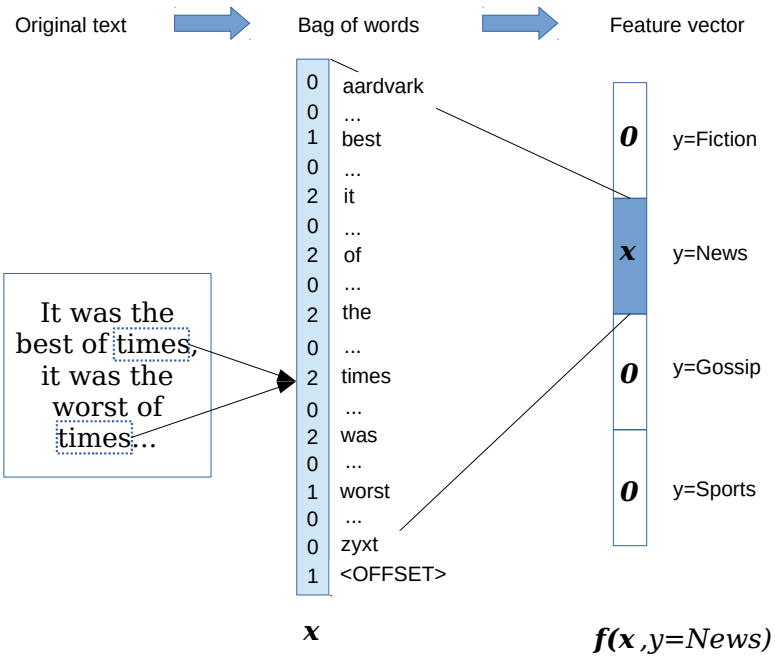

Figure 2.1: The bag-of-words and feature vector representations, for a hypothetical text classification task.

associated dictionary. For example,[3]

$$\theta_{(E,bicycle)} = 1 \qquad \theta_{(S,bicycle)} = 0$$
$$\theta_{(E,bicicleta)} = 0 \qquad \theta_{(S,bicicleta)} = 1$$
$$\theta_{(E,con)} = 1 \qquad \theta_{(S,con)} = 1$$
$$\theta_{(E,ordinateur)} = 0 \qquad \theta_{(S,ordinateur)} = 0.$$

Similarly, if we want to distinguish positive and negative sentiment, we could use positive and negative **sentiment lexicons** (see § 4.1.2), which are defined by social psychologists (Tausczik and Pennebaker, 2010).

But it is usually not easy to set classification weights by hand, due to the large number of words and the difficulty of selecting exact numerical weights. Instead, we will learn the weights from data. Email users manually label messages as SPAM; newspapers label their own articles as BUSINESS or STYLE. Using such **instance labels**, we can automatically acquire weights using **supervised machine learning**. This chapter will discuss several machine learning approaches for classification. The first is based on probability. For a review of probability, consult Appendix A.

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

2.2 Naïve Bayes

The **joint probability** of a bag of words $\boldsymbol{x}$ and its true label y is written $\text{p}(\boldsymbol{x}, y)$. Suppose we have a dataset of N labeled instances, $\{(\boldsymbol{x}^{(i)}, y^{(i)})\}_{i=1}^{N}$, which we assume are **independent and identically distributed (IID)** (see § A.3). Then the joint probability of the entire dataset, written $\text{p}(\boldsymbol{x}^{(1:N)}, y^{(1:N)})$, is equal to $\prod_{i=1}^{N} \text{p}_{X,Y}(\boldsymbol{x}^{(i)}, y^{(i)})$.[4]

What does this have to do with classification? One approach to classification is to set the weights $\boldsymbol{\theta}$ so as to maximize the joint probability of a **training set** of labeled documents. This is known as **maximum likelihood estimation**:

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\text{argmax}}\, \text{p}(\boldsymbol{x}^{(1:N)}, y^{(1:N)}; \boldsymbol{\theta}) \quad [2.8]$$

$$= \underset{\boldsymbol{\theta}}{\text{argmax}} \prod_{i=1}^{N} \text{p}(\boldsymbol{x}^{(i)}, y^{(i)}; \boldsymbol{\theta}) \quad [2.9]$$

$$= \underset{\boldsymbol{\theta}}{\text{argmax}} \sum_{i=1}^{N} \log \text{p}(\boldsymbol{x}^{(i)}, y^{(i)}; \boldsymbol{\theta}). \quad [2.10]$$

The notation $\text{p}(\boldsymbol{x}^{(i)}, y^{(i)}; \boldsymbol{\theta})$ indicates that $\boldsymbol{\theta}$ is a *parameter* of the probability function. The product of probabilities can be replaced by a sum of log-probabilities because the $\log$ function is monotonically increasing over positive arguments, and so the same $\boldsymbol{\theta}$ will maximize both the probability and its logarithm. Working with logarithms is desirable because of numerical stability: on a large dataset, multiplying many probabilities can **underflow** to zero.[5]

The probability $\text{p}(\boldsymbol{x}^{(i)}, y^{(i)}; \boldsymbol{\theta})$ is defined through a **generative model** — an idealized random process that has generated the observed data.[6] Algorithm 1 describes the generative model underlying the **Naïve Bayes** classifier, with parameters $\boldsymbol{\theta} = \{\boldsymbol{\mu}, \boldsymbol{\phi}\}$.

- The first line of this generative model encodes the assumption that the instances are mutually independent: neither the label nor the text of document i affects the label or text of document j.[7] Furthermore, the instances are identically distributed: the

[4] The notation $\text{p}_{X,Y}(\boldsymbol{x}^{(i)}, y^{(i)})$ indicates the joint probability that random variables X and Y take the specific values $\boldsymbol{x}^{(i)}$ and $y^{(i)}$ respectively. The subscript will often be omitted when it is clear from context. For a review of random variables, see Appendix A.

[5] Throughout this text, you may assume all logarithms and exponents are base 2, unless otherwise indicated. Any reasonable base will yield an identical classifier, and base 2 is most convenient for working out examples by hand.

[6] Generative models will be used throughout this text. They explicitly define the assumptions underlying the form of a probability distribution over observed and latent variables. For a readable introduction to generative models in statistics, see Blei (2014).

[7] Can you think of any cases in which this assumption is too strong?

Algorithm 1 Generative process for the Naïve Bayes classification model

for Instance $i \in \{1, 2, \ldots, N\}$ **do**:
  Draw the label $y^{(i)} \sim \text{Categorical}(\boldsymbol{\mu})$;
  Draw the word counts $\boldsymbol{x}^{(i)} \mid y^{(i)} \sim \text{Multinomial}(\boldsymbol{\phi}_{y^{(i)}})$.

distributions over the label $y^{(i)}$ and the text $\boldsymbol{x}^{(i)}$ (conditioned on $y^{(i)}$) are the same for all instances i. In other words, we make the assumption that every document has the same distribution over labels, and that each document's distribution over words depends only on the label, and not on anything else about the document. We also assume that the documents don't affect each other: if the word *whale* appears in document $i = 7$, that does not make it any more or less likely that it will appear again in document $i = 8$.

- The second line of the generative model states that the random variable $y^{(i)}$ is drawn from a categorical distribution with parameter $\boldsymbol{\mu}$. Categorical distributions are like weighted dice: the column vector $\boldsymbol{\mu} = [\mu_1; \mu_2; \ldots; \mu_K]$ gives the probabilities of each label, so that the probability of drawing label y is equal to μ_y. For example, if $\mathcal{Y} = \{\text{POSITIVE}, \text{NEGATIVE}, \text{NEUTRAL}\}$, we might have $\boldsymbol{\mu} = [0.1; 0.7; 0.2]$. We require $\sum_{y \in \mathcal{Y}} \mu_y = 1$ and $\mu_y \geq 0$, $\forall y \in \mathcal{Y}$: each label's probability is non-negative, and the sum of these probabilities is equal to one. [8]
- The third line describes how the bag-of-words counts $\boldsymbol{x}^{(i)}$ are generated. By writing $\boldsymbol{x}^{(i)} \mid y^{(i)}$, this line indicates that the word counts are conditioned on the label, so that the joint probability is factored using the chain rule,

$$\text{p}_{X,Y}(\boldsymbol{x}^{(i)}, y^{(i)}) = \text{p}_{X|Y}(\boldsymbol{x}^{(i)} \mid y^{(i)}) \times \text{p}_Y(y^{(i)}). \quad [2.11]$$

  The specific distribution $\text{p}_{X|Y}$ is the **multinomial**, which is a probability distribution over vectors of non-negative counts. The probability mass function for this distribution is:

$$\text{p}_{\text{mult}}(\boldsymbol{x}; \boldsymbol{\phi}) = B(\boldsymbol{x}) \prod_{j=1}^{V} \phi_j^{x_j} \quad [2.12]$$

$$B(\boldsymbol{x}) = \frac{\left(\sum_{j=1}^{V} x_j\right)!}{\prod_{j=1}^{V} (x_j!)}. \quad [2.13]$$

[8] Formally, we require $\boldsymbol{\mu} \in \Delta^{K-1}$, where Δ^{K-1} is the $K-1$ **probability simplex**, the set of all vectors of K nonnegative numbers that sum to one. Because of the sum-to-one constraint, there are $K-1$ degrees of freedom for a vector of size K.

As in the categorical distribution, the parameter ϕ_j can be interpreted as a probability: specifically, the probability that any given token in the document is the word j. The multinomial distribution involves a product over words, with each term in the product equal to the probability ϕ_j, exponentiated by the count x_j. Words that have zero count play no role in this product, because $\phi_j^0 = 1$. The term $B(\boldsymbol{x})$ is called the **multinomial coefficient**. It doesn't depend on $\boldsymbol{\phi}$, and can usually be ignored. Can you see why we need this term at all?[9]

The notation $\text{p}(\boldsymbol{x} \mid y; \boldsymbol{\phi})$ indicates the conditional probability of word counts $\boldsymbol{x}$ given label y, with parameter $\boldsymbol{\phi}$, which is equal to $\text{p}_{\text{mult}}(\boldsymbol{x}; \boldsymbol{\phi}_y)$. By specifying the multinomial distribution, we describe the **multinomial Naïve Bayes** classifier. Why "naïve"? Because the multinomial distribution treats each word token independently, conditioned on the class: the probability mass function factorizes across the counts.[10]

2.2.1 Types and tokens

A slight modification to the generative model of Naïve Bayes is shown in Algorithm 2. Instead of generating a vector of counts of **types**, $\boldsymbol{x}$, this model generates a *sequence* of **tokens**, $\boldsymbol{w} = (w_1, w_2, \ldots, w_M)$. The distinction between types and tokens is critical: $x_j \in \{0, 1, 2, \ldots, M\}$ is the count of word type j in the vocabulary, e.g., the number of times the word *cannibal* appears; $w_m \in \mathcal{V}$ is the identity of token m in the document, e.g. $w_m =$ *cannibal*.

The probability of the sequence $\boldsymbol{w}$ is a product of categorical probabilities. Algorithm 2 makes a conditional independence assumption: each token $w_m^{(i)}$ is independent of all other tokens $w_{n \neq m}^{(i)}$, conditioned on the label $y^{(i)}$. This is identical to the "naïve" independence assumption implied by the multinomial distribution, and as a result, the optimal parameters for this model are identical to those in multinomial Naïve Bayes. For any instance, the probability assigned by this model is proportional to the probability under multinomial Naïve Bayes. The constant of proportionality is the multinomial coefficient $B(\boldsymbol{x})$. Because $B(\boldsymbol{x}) \geq 1$, the probability for a vector of counts $\boldsymbol{x}$ is at least as large as the probability for a list of words $\boldsymbol{w}$ that induces the same counts: there can be many word sequences that correspond to a single vector of counts. For example, *man bites dog* and *dog bites man* correspond to an identical count vector, $\{\textit{bites} : 1, \textit{dog} : 1, \textit{man} : 1\}$, and $B(\boldsymbol{x})$ is equal to the total number of possible word orderings for count vector $\boldsymbol{x}$.

[9] Technically, a multinomial distribution requires a second parameter, the total number of word counts in $\boldsymbol{x}$. In the bag-of-words representation is equal to the number of words in the document. However, this parameter is irrelevant for classification.

[10] 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 the features are conditionally independent, given the label. For example, a multivariate Gaussian with diagonal covariance is naïve in exactly the same sense.

Algorithm 2 Alternative generative process for the Naïve Bayes classification model

for Instance $i \in \{1, 2, \ldots, N\}$ **do**:
  Draw the label $y^{(i)} \sim \text{Categorical}(\boldsymbol{\mu})$;
  for Token $m \in \{1, 2, \ldots, M_i\}$ **do**:
    Draw the token $w_m^{(i)} \mid y^{(i)} \sim \text{Categorical}(\boldsymbol{\phi}_{y^{(i)}})$.

Sometimes it is useful to think of instances as counts of types, $\boldsymbol{x}$; other times, it is better to think of them as sequences of tokens, $\boldsymbol{w}$. If the tokens are generated from a model that assumes conditional independence, then these two views lead to probability models that are identical, except for a scaling factor that does not depend on the label or the parameters.

2.2.2 Prediction

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

$$\hat{y} = \underset{y}{\text{argmax}} \log \text{p}(\boldsymbol{x}, y; \boldsymbol{\mu}, \boldsymbol{\phi}) \quad [2.14]$$

$$= \underset{y}{\text{argmax}} \log \text{p}(\boldsymbol{x} \mid y; \boldsymbol{\phi}) + \log \text{p}(y; \boldsymbol{\mu}) \quad [2.15]$$

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

$$\log \text{p}(\boldsymbol{x} \mid y; \boldsymbol{\phi}) + \log \text{p}(y; \boldsymbol{\mu}) = \log \left[B(\boldsymbol{x}) \prod_{j=1}^{V} \phi_{y,j}^{x_j} \right] + \log \mu_y \quad [2.16]$$

$$= \log B(\boldsymbol{x}) + \sum_{j=1}^{V} x_j \log \phi_{y,j} + \log \mu_y \quad [2.17]$$

$$= \log B(\boldsymbol{x}) + \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y), \quad [2.18]$$

where

$$\boldsymbol{\theta} = [\boldsymbol{\theta}^{(1)}; \boldsymbol{\theta}^{(2)}; \ldots; \boldsymbol{\theta}^{(K)}] \quad [2.19]$$

$$\boldsymbol{\theta}^{(y)} = [\log \phi_{y,1}; \log \phi_{y,2}; \ldots; \log \phi_{y,V}; \log \mu_y] \quad [2.20]$$

The feature function $\boldsymbol{f}(\boldsymbol{x}, y)$ is a vector of V word counts and an offset, padded by zeros for the labels not equal to y (see Equations 2.3-2.5, and Figure 2.1). This construction ensures that the inner product $\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y)$ only activates the features whose weights are in $\boldsymbol{\theta}^{(y)}$. These features and weights are all we need to compute the joint log-probability $\log \text{p}(\boldsymbol{x}, y)$ for each y. This is a key point: through this notation, we have converted the problem of computing the log-likelihood for a document-label pair $(\boldsymbol{x}, y)$ into the computation of a vector inner product.

2.2.3 Estimation

The parameters of the categorical and multinomial distributions have a simple interpretation: they are vectors of expected frequencies for each possible event. Based on this interpretation, it is tempting to set the parameters empirically,

$$\phi_{y,j} = \frac{\text{count}(y, j)}{\sum_{j'=1}^{V} \text{count}(y, j')} = \frac{\sum_{i:y^{(i)}=y} x_j^{(i)}}{\sum_{j'=1}^{V} \sum_{i:y^{(i)}=y} x_{j'}^{(i)}}, \quad [2.21]$$

where $\text{count}(y, j)$ refers to the count of word j in documents with label y.

Equation 2.21 defines the **relative frequency estimate** for $\boldsymbol{\phi}$. It can be justified as a **maximum likelihood estimate**: the estimate that maximizes the probability $\text{p}(\boldsymbol{x}^{(1:N)}, y^{(1:N)}; \boldsymbol{\theta})$. Based on the generative model in Algorithm 1, the log-likelihood is,

$$\mathcal{L}(\boldsymbol{\phi}, \boldsymbol{\mu}) = \sum_{i=1}^{N} \log \text{p}_{\text{mult}}(\boldsymbol{x}^{(i)}; \boldsymbol{\phi}_{y^{(i)}}) + \log \text{p}_{\text{cat}}(y^{(i)}; \boldsymbol{\mu}), \quad [2.22]$$

which is now written as a function $\mathcal{L}$ of the parameters $\boldsymbol{\phi}$ and $\boldsymbol{\mu}$. Let's continue to focus on the parameters $\boldsymbol{\phi}$. Since $\text{p}(y)$ is constant with respect to $\boldsymbol{\phi}$, we can drop it:

$$\mathcal{L}(\boldsymbol{\phi}) = \sum_{i=1}^{N} \log \text{p}_{\text{mult}}(\boldsymbol{x}^{(i)}; \boldsymbol{\phi}_{y^{(i)}}) = \sum_{i=1}^{N} \log B(\boldsymbol{x}^{(i)}) + \sum_{j=1}^{V} x_j^{(i)} \log \phi_{y^{(i)},j}, \quad [2.23]$$

where $B(\boldsymbol{x}^{(i)})$ is constant with respect to $\boldsymbol{\phi}$.

Maximum-likelihood estimation chooses $\boldsymbol{\phi}$ to maximize the log-likelihood $\mathcal{L}$. However, the solution must obey the following constraints:

$$\sum_{j=1}^{V} \phi_{y,j} = 1 \quad \forall y \quad [2.24]$$

These constraints can be incorporated by adding a set of Lagrange multipliers to the objective (see Appendix B for more details). To solve for each $\boldsymbol{\theta}_y$, we maximize the Lagrangian,

$$\ell(\boldsymbol{\phi}_y) = \sum_{i:y^{(i)}=y} \sum_{j=1}^{V} x_j^{(i)} \log \phi_{y,j} - \lambda(\sum_{j=1}^{V} \phi_{y,j} - 1). \quad [2.25]$$

Differentiating with respect to the parameter $\phi_{y,j}$ yields,

$$\frac{\partial \ell(\boldsymbol{\phi}_y)}{\partial \phi_{y,j}} = \sum_{i:y^{(i)}=y} x_j^{(i)} / \phi_{y,j} - \lambda. \quad [2.26]$$

The solution is obtained by setting each element in this vector of derivatives equal to zero,

$$\lambda\phi_{y,j} = \sum_{i:y^{(i)}=y} x_j^{(i)} \quad [2.27]$$

$$\phi_{y,j} \propto \sum_{i:y^{(i)}=y} x_j^{(i)} = \sum_{i=1}^{N} \delta\left(y^{(i)} = y\right) x_j^{(i)} = \text{count}(y, j), \quad [2.28]$$

where $\delta\left(y^{(i)} = y\right)$ is a **delta function**, also sometimes called an indicator function, which returns one if $y^{(i)} = y$. The symbol $\propto$ indicates that $\phi_{y,j}$ is **proportional to** the right-hand side of the equation.

Equation 2.28 shows three different notations for the same thing: a sum over the word counts for all documents i such that the label $y^{(i)} = y$. This gives a solution for each $\boldsymbol{\phi}_y$ up to a constant of proportionality. Now recall the constraint $\sum_{j=1}^{V} \boldsymbol{\phi}_{y,j} = 1$, which arises because $\boldsymbol{\phi}_y$ represents a vector of probabilities for each word in the vocabulary. This constraint leads to an exact solution, which does not depend on λ:

$$\phi_{y,j} = \frac{\text{count}(y, j)}{\sum_{j'=1}^{V} \text{count}(y, j')}. \quad [2.29]$$

This is equal to the relative frequency estimator from Equation 2.21. A similar derivation gives $\mu_y \propto \sum_{i=1}^{N} \delta\left(y^{(i)} = y\right)$.

2.2.4 Smoothing

With text data, there are likely to be pairs of labels and words that never appear in the training set, leaving $\phi_{y,j} = 0$. For example, the word *molybdenum* may have never yet appeared in a work of fiction. But choosing a value of $\phi_{\text{FICTION},\textit{molybdenum}} = 0$ would allow this single feature to completely veto a label, since $\text{p}(\text{FICTION} \mid \boldsymbol{x}) = 0$ if $\boldsymbol{x}_{\textit{molybdenum}} > 0$.

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

$$\phi_{y,j} = \frac{\alpha + \text{count}(y, j)}{V\alpha + \sum_{j'=1}^{V} \text{count}(y, j')} \quad [2.30]$$

This is called **Laplace smoothing**.[11] The pseudocount α is a **hyperparameter**, because it controls the form of the log-likelihood function, which in turn drives the estimation of $\boldsymbol{\phi}$.

[11] Laplace smoothing has a Bayesian justification, in which the generative model is extended to include $\boldsymbol{\phi}$ as a random variable. The resulting distribution over $\boldsymbol{\phi}$ depends on both the data ($\boldsymbol{x}$ and y) and the **prior probability** $\text{p}(\boldsymbol{\phi}; \alpha)$. The corresponding estimate of $\boldsymbol{\phi}$ is called **maximum a posteriori**, or MAP. This is in contrast with maximum likelihood, which depends only on the data.

Smoothing reduces variance, but moves us away from the maximum likelihood estimate: it imposes a **bias**. In this case, the bias points towards uniform probabilities. Machine learning theory shows that errors on heldout data can be attributed to the sum of bias and variance (Mohri et al., 2012). In general, techniques for reducing variance often increase the bias, leading to a **bias-variance tradeoff**.

- Unbiased classifiers may **overfit** the training data, yielding poor performance on unseen data.
- But if the smoothing is too large, the resulting classifier can **underfit** instead. In the limit of $\alpha \rightarrow \infty$, there is zero variance: you get the same classifier, regardless of the data. However, the bias is likely to be large.

Similar issues arise throughout machine learning. Later in this chapter we will encounter **regularization**, which controls the bias-variance tradeoff for logistic regression and large-margin classifiers (§ 2.5.1); § 3.3.2 describes techniques for controlling variance in deep learning; chapter 6 describes more elaborate methods for smoothing empirical probabilities.

2.2.5 Setting hyperparameters

Returning to Naïve Bayes, how should we choose the best value of hyperparameters like α? Maximum likelihood will not work: the maximum likelihood estimate of α on the training set will always be $\alpha = 0$. In many cases, what we really want is **accuracy**: the number of correct predictions, divided by the total number of predictions. (Other measures of classification performance are discussed in § 4.4.) As we will see, it is hard to optimize for accuracy directly. But for scalar hyperparameters like α, tuning can be performed by a simple heuristic called **grid search**: try a set of values (e.g., $\alpha \in \{0.001, 0.01, 0.1, 1, 10\}$), compute the accuracy for each value, and choose the setting that maximizes the accuracy.

The goal is to tune α so that the classifier performs well on *unseen* data. For this reason, the data used for hyperparameter tuning should not overlap the training set, where very small values of α will be preferred. Instead, we hold out a **development set** (also called a **tuning set**) for hyperparameter selection. This development set may consist of a small fraction of the labeled data, such as 10%.

We also want to predict the performance of our classifier on unseen data. To do this, we must hold out a separate subset of data, called the **test set**. It is critical that the test set not overlap with either the training or development sets, or else we will overestimate the performance that the classifier will achieve on unlabeled data in the future. The test set should also not be used when making modeling decisions, such as the form of the feature function, the size of the vocabulary, and so on (these decisions are reviewed in chapter 4.) The ideal practice is to use the test set only once — otherwise, the test set is used to guide

the classifier design, and test set accuracy will diverge from accuracy on truly unseen data. Because annotated data is expensive, this ideal can be hard to follow in practice, and many test sets have been used for decades. But in some high-impact applications like machine translation and information extraction, new test sets are released every year.

When only a small amount of labeled data is available, the test set accuracy can be unreliable. K-fold **cross-validation** is one way to cope with this scenario: the labeled data is divided into K folds, and each fold acts as the test set, while training on the other folds. The test set accuracies are then aggregated. In the extreme, each fold is a single data point; this is called **leave-one-out cross-validation**. To perform hyperparameter tuning in the context of cross-validation, another fold can be used for grid search. It is important not to repeatedly evaluate the cross-validated accuracy while making design decisions about the classifier, or you will overstate the accuracy on truly unseen data.

2.3 Discriminative learning

Naïve Bayes is easy to work with: the weights can be estimated in closed form, and the probabilistic interpretation makes it relatively easy to extend. However, the assumption that features are independent can seriously limit its accuracy. Thus far, we have defined the feature function $\boldsymbol{f}(\boldsymbol{x}, y)$ so that it corresponds to bag-of-words features: one feature per word in the vocabulary. In natural language, bag-of-words features violate the assumption of conditional independence — for example, the probability that a document will contain the word *naïve* is surely higher given that it also contains the word *Bayes* — but this violation is relatively mild.

However, good performance on text classification often requires features that are richer than the bag-of-words:

- To better handle out-of-vocabulary terms, we want features that apply to multiple words, such as prefixes and suffixes (e.g., *anti-*, *un-*, *-ing*) and capitalization.
- We also want n**-gram** features that apply to multi-word units: **bigrams** (e.g., *not good*, *not bad*), **trigrams** (e.g., *not so bad*, *lacking any decency*, *never before imagined*), and beyond.

These features flagrantly violate the Naïve Bayes independence assumption. Consider what happens if we add a prefix feature. Under the Naïve Bayes assumption, the joint probability of a word and its prefix are computed with the following approximation:[12]

$$\Pr(\text{word} = \textit{unfit}, \text{prefix} = \textit{un-} \mid y) \approx \Pr(\text{prefix} = \textit{un-} \mid y) \times \Pr(\text{word} = \textit{unfit} \mid y).$$

[12]The notation $\Pr(\cdot)$ refers to the probability of an event, and $\mathrm{p}(\cdot)$ refers to the probability density or mass for a random variable (see Appendix A).

To test the quality of the approximation, we can manipulate the left-hand side by applying the chain rule,

$$\Pr(\text{word} = \textit{unfit}, \text{prefix} = \textit{un-} \mid y) = \Pr(\text{prefix} = \textit{un-} \mid \text{word} = \textit{unfit}, y) \quad [2.31]$$
$$\times \Pr(\text{word} = \textit{unfit} \mid y) \quad [2.32]$$

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

$$\Pr(\text{word} = \textit{unfit}, \text{prefix} = \textit{un-} \mid y) = 1 \times \Pr(\text{word} = \textit{unfit} \mid y) \quad [2.33]$$
$$\gg \Pr(\text{prefix} = \textit{un-} \mid y) \times \Pr(\text{word} = \textit{unfit} \mid y), \quad [2.34]$$

because the probability of any given word starting with the prefix *un-* is much less than one. Naïve Bayes will systematically underestimate the true probabilities of conjunctions of positively correlated features. To use such features, we need learning algorithms that do not rely on an independence assumption.

The origin of the Naïve Bayes independence assumption is the learning objective, $\text{p}(\boldsymbol{x}^{(1:N)}, y^{(1:N)})$, which requires modeling the probability of the observed text. In classification problems, we are always given $\boldsymbol{x}$, and are only interested in predicting the label y. In this setting, modeling the probability of the text $\boldsymbol{x}$ seems like a difficult and unnecessary task. **Discriminative learning** algorithms avoid this task, and focus directly on the problem of predicting y.

2.3.1 Perceptron

In Naïve Bayes, the weights can be interpreted as parameters of a probabilistic model. But this model requires an independence assumption that usually does not hold, and limits our choice of features. Why not forget about probability and learn the weights in an error-driven way? The **perceptron** algorithm, shown in Algorithm 3, is one way to do this.

The algorithm is simple: if you make a mistake, increase the weights for features that are active with the correct label $y^{(i)}$, and decrease the weights for features that are active with the guessed label $\hat{y}$. Perceptron is an **online learning** algorithm, since the classifier weights change after every example. This is different from Naïve Bayes, which is a **batch learning** algorithm: it computes statistics over the entire dataset, and then sets the weights in a single operation. Algorithm 3 is vague about when this online learning procedure terminates. We will return to this issue shortly.

The perceptron algorithm may seem like an unprincipled heuristic: Naïve Bayes has a solid foundation in probability, but the perceptron is just adding and subtracting constants from the weights every time there is a mistake. Will this really work? In fact, there is some nice theory for the perceptron, based on the concept of **linear separability**. Informally, a dataset with binary labels ($y \in \{0, 1\}$) is linearly separable if it is possible to draw a

Algorithm 3 Perceptron learning algorithm

```
procedure PERCEPTRON(x^(1:N), y^(1:N))
    t ← 0
    θ^(0) ← 0
    repeat
        t ← t + 1
        Select an instance i
        ŷ ← argmax_y θ^(t−1) · f(x^(i), y)
        if ŷ ≠ y^(i) then
            θ^(t) ← θ^(t−1) + f(x^(i), y^(i)) − f(x^(i), ŷ)
        else
            θ^(t) ← θ^(t−1)
    until tired
    return θ^(t)
```

hyperplane (a line in many dimensions), such that on each side of the hyperplane, all instances have the same label. This definition can be formalized and extended to multiple labels:

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

$$\exists \boldsymbol{\theta}, \rho > 0 : \forall (\boldsymbol{x}^{(i)}, y^{(i)}) \in \mathcal{D}, \quad \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) \geq \rho + \max_{y' \neq y^{(i)}} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y'). \tag{2.35}$$

Linear separability is important because of the following guarantee: if your data is linearly separable, then the perceptron algorithm will find a separator (Novikoff, 1962).[13] So while the perceptron may seem heuristic, it is guaranteed to succeed, if the learning problem is easy enough.

How useful is this proof? Minsky and Papert (1969) famously proved that the simple logical function of *exclusive-or* is not separable, and that a perceptron is therefore incapable of learning this function. But this is not just an issue for the perceptron: any linear classification algorithm, including Naïve Bayes, will fail on this task. Text classification problems usually involve high dimensional feature spaces, with thousands or millions of

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

features. For these problems, it is very likely that the training data is indeed separable. And even if the dataset is not separable, it is still possible to place an upper bound on the number of errors that the perceptron algorithm will make (Freund and Schapire, 1999).

2.3.2 Averaged perceptron

The perceptron iterates over the data repeatedly — until "tired", as described in Algorithm 3. If the data is linearly separable, the perceptron will eventually find a separator, and we can stop once all training instances are classified correctly. But if the data is not linearly separable, the perceptron can *thrash* between two or more weight settings, never converging. In this case, how do we know that we can stop training, and how should we choose the final weights? An effective practical solution is to *average* the perceptron weights across all iterations.

This procedure is shown in Algorithm 4. The learning algorithm is nearly identical, but we also maintain a vector of the sum of the weights, $\boldsymbol{m}$. At the end of the learning procedure, we divide this sum by the total number of updates t, to compute the average weights, $\overline{\boldsymbol{\theta}}$. These average weights are then used for prediction. In the algorithm sketch, the average is computed from a running sum, $\boldsymbol{m} \leftarrow \boldsymbol{m} + \boldsymbol{\theta}$. However, this is inefficient, because it requires $|\boldsymbol{\theta}|$ operations to update the running sum. When $\boldsymbol{f}(\boldsymbol{x}, y)$ is sparse, $|\boldsymbol{\theta}| \gg |\boldsymbol{f}(\boldsymbol{x}, y)|$ for any individual $(\boldsymbol{x}, y)$. This means that computing the running sum will be much more expensive than computing of the update to $\boldsymbol{\theta}$ itself, which requires only $2 \times |\boldsymbol{f}(\boldsymbol{x}, y)|$ operations. One of the exercises is to sketch a more efficient algorithm for computing the averaged weights.

Even if the dataset is not separable, the averaged weights will eventually converge. One possible stopping criterion is to check the difference between the average weight vectors after each pass through the data: if the norm of the difference falls below some predefined threshold, we can stop training. Another stopping criterion is to hold out some data, and to measure the predictive accuracy on this heldout data. When the accuracy on the heldout data starts to decrease, the learning algorithm has begun to **overfit** the training set. At this point, it is probably best to stop; this stopping criterion is known as **early stopping**.

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

2.4 Loss functions and large-margin classification

Naïve Bayes chooses the weights $\boldsymbol{\theta}$ by maximizing the joint log-likelihood $\log \mathrm{p}(\boldsymbol{x}^{(1:N)}, y^{(1:N)})$. By convention, optimization problems are generally formulated as minimization of a **loss function**. The input to a loss function is the vector of weights $\boldsymbol{\theta}$, and the output is a

Algorithm 4 Averaged perceptron learning algorithm

procedure AVG-PERCEPTRON($\boldsymbol{x}^{(1:N)}, \boldsymbol{y}^{(1:N)}$)
  $t \leftarrow 0$
  $\boldsymbol{\theta}^{(0)} \leftarrow 0$
  repeat
    $t \leftarrow t + 1$
    Select an instance i
    $\hat{y} \leftarrow \text{argmax}_y \boldsymbol{\theta}^{(t-1)} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y)$
    if $\hat{y} \neq y^{(i)}$ **then**
      $\boldsymbol{\theta}^{(t)} \leftarrow \boldsymbol{\theta}^{(t-1)} + \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \boldsymbol{f}(\boldsymbol{x}^{(i)}, \hat{y})$
    else
      $\boldsymbol{\theta}^{(t)} \leftarrow \boldsymbol{\theta}^{(t-1)}$
    $\boldsymbol{m} \leftarrow \boldsymbol{m} + \boldsymbol{\theta}^{(t)}$
  until tired
  $\overline{\boldsymbol{\theta}} \leftarrow \frac{1}{t}\boldsymbol{m}$
  return $\overline{\boldsymbol{\theta}}$

non-negative number, measuring the performance of the classifier on a training instance. Formally, the loss $\ell(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)})$ is then a measure of the performance of the weights $\boldsymbol{\theta}$ on the instance $(\boldsymbol{x}^{(i)}, y^{(i)})$. The goal of learning is to minimize the sum of the losses across all instances in the training set.

We can trivially reformulate maximum likelihood as a loss function, by defining the loss function to be the *negative* log-likelihood:

$$\log \text{p}(\boldsymbol{x}^{(1:N)}, y^{(1:N)}; \boldsymbol{\theta}) = \sum_{i=1}^{N} \log \text{p}(\boldsymbol{x}^{(i)}, y^{(i)}; \boldsymbol{\theta}) \quad [2.36]$$

$$\ell_{\text{NB}}(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) = -\log \text{p}(\boldsymbol{x}^{(i)}, y^{(i)}; \boldsymbol{\theta}) \quad [2.37]$$

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\text{argmin}} \sum_{i=1}^{N} \ell_{\text{NB}}(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) \quad [2.38]$$

$$= \underset{\boldsymbol{\theta}}{\text{argmax}} \sum_{i=1}^{N} \log \text{p}(\boldsymbol{x}^{(i)}, y^{(i)}; \boldsymbol{\theta}). \quad [2.39]$$

The problem of minimizing ℓ_{NB} is thus identical to maximum-likelihood estimation.

Loss functions provide a general framework for comparing learning objectives. For example, an alternative loss function is the **zero-one loss**,

$$\ell_{0\text{-}1}(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) = \begin{cases} 0, & y^{(i)} = \text{argmax}_y \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y) \\ 1, & \text{otherwise} \end{cases} \quad [2.40]$$

The zero-one loss is zero if the instance is correctly classified, and one otherwise. The sum of zero-one losses is proportional to the error rate of the classifier on the training data. Since a low error rate is often the ultimate goal of classification, this may seem ideal. But the zero-one loss has several problems. One is that it is **non-convex**,[14] which means that there is no guarantee that gradient-based optimization will be effective. A more serious problem is that the derivatives are useless: the partial derivative with respect to any parameter is zero everywhere, except at the points where $\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y) = \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, \hat{y})$ for some $\hat{y}$. At those points, the loss is discontinuous, and the derivative is undefined.

The perceptron optimizes a loss function that has better properties for learning:

$$\ell_{\text{PERCEPTRON}}(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) = \max_{y \in \mathcal{Y}} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y) - \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}), \quad [2.41]$$

When $\hat{y} = y^{(i)}$, the loss is zero; otherwise, it increases linearly with the gap between the score for the predicted label $\hat{y}$ and the score for the true label $y^{(i)}$. Plotting this loss against the input $\max_{y \in \mathcal{Y}} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y) - \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)})$ gives a hinge shape, motivating the name **hinge loss**.

To see why this is the loss function optimized by the perceptron, take the derivative with respect to $\boldsymbol{\theta}$,

$$\frac{\partial}{\partial \boldsymbol{\theta}} \ell_{\text{PERCEPTRON}}(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) = \boldsymbol{f}(\boldsymbol{x}^{(i)}, \hat{y}) - \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}). \quad [2.42]$$

At each instance, the perceptron algorithm takes a step of magnitude one in the opposite direction of this **gradient**, $\nabla_{\boldsymbol{\theta}} \ell_{\text{PERCEPTRON}} = \frac{\partial}{\partial \boldsymbol{\theta}} \ell_{\text{PERCEPTRON}}(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)})$. As we will see in § 2.6, this is an example of the optimization algorithm **stochastic gradient descent**, applied to the objective in Equation 2.41.

***Breaking ties with subgradient descent** [15] Careful readers will notice the tacit assumption that there is a unique $\hat{y}$ that maximizes $\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y)$. What if there are two or more labels that maximize this function? Consider binary classification: if the maximizer is $y^{(i)}$, then the gradient is zero, and so is the perceptron update; if the maximizer is $\hat{y} \neq y^{(i)}$, then the update is the difference $\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \boldsymbol{f}(\boldsymbol{x}^{(i)}, \hat{y})$. The underlying issue is that the perceptron loss is not **smooth**, because the first derivative has a discontinuity at the hinge point, where the score for the true label $y^{(i)}$ is equal to the score for some other label $\hat{y}$. At this point, there is no unique gradient; rather, there is a set of **subgradients**. A vector $\boldsymbol{v}$ is

[14] 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. In words, any weighted average of the output of f applied to any two points is larger than the output of f when applied to the weighted average of the same two points. Convexity implies that any local minimum is also a global minimum, and there are many effective techniques for optimizing convex functions (Boyd and Vandenberghe, 2004). See Appendix B for a brief review.

[15] Throughout this text, advanced topics will be marked with an asterisk.

a subgradient of the function g at $\boldsymbol{u}_0$ iff $g(\boldsymbol{u}) - g(\boldsymbol{u}_0) \geq \boldsymbol{v} \cdot (\boldsymbol{u} - \boldsymbol{u}_0)$ for all $\boldsymbol{u}$. Graphically, this defines the set of hyperplanes that include $g(\boldsymbol{u}_0)$ and do not intersect g at any other point. As we approach the hinge point from the left, the gradient is $\boldsymbol{f}(\boldsymbol{x}, \hat{y}) - \boldsymbol{f}(\boldsymbol{x}, y)$; as we approach from the right, the gradient is $\mathbf{0}$. At the hinge point, the subgradients include all vectors that are bounded by these two extremes. In subgradient descent, *any* subgradient can be used (Bertsekas, 2012). Since both $\mathbf{0}$ and $\boldsymbol{f}(\boldsymbol{x}, \hat{y}) - \boldsymbol{f}(\boldsymbol{x}, y)$ are subgradients at the hinge point, either one can be used in the perceptron update. This means that if multiple labels maximize $\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y)$, any of them can be used in the perceptron update.

Perceptron versus Naïve Bayes The perceptron loss function has some pros and cons with respect to the negative log-likelihood loss implied by Naïve Bayes.

- Both ℓ_{NB} and $\ell_{\text{PERCEPTRON}}$ are convex, making them relatively easy to optimize. However, ℓ_{NB} can be optimized in closed form, while $\ell_{\text{PERCEPTRON}}$ requires iterating over the dataset multiple times.
- ℓ_{NB} can suffer *infinite* loss on a single example, since the logarithm of zero probability is negative infinity. Naïve Bayes will therefore overemphasize some examples, and underemphasize others.
- The Naïve Bayes classifier assumes that the observed features are conditionally independent, given the label, and the performance of the classifier depends on the extent to which this assumption holds. The perceptron requires no such assumption.
- $\ell_{\text{PERCEPTRON}}$ treats all correct answers equally. Even if $\boldsymbol{\theta}$ only gives the correct answer by a tiny margin, the loss is still zero.

2.4.1 Online large margin classification

This last comment suggests a potential problem with the perceptron. Suppose a test example is very close to a training example, but not identical. If the classifier only gets the correct answer on the training example by a small amount, then it may give a different answer on the nearby test instance. To formalize this intuition, define the **margin** as,

$$\gamma(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) = \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \max_{y \neq y^{(i)}} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y). \quad [2.43]$$

The margin represents the difference between the score for the correct label $y^{(i)}$, and the score for the highest-scoring incorrect label. The intuition behind **large margin classification** is that it is not enough to label the training data correctly — the correct label should be separated from other labels by a comfortable margin. This idea can be encoded

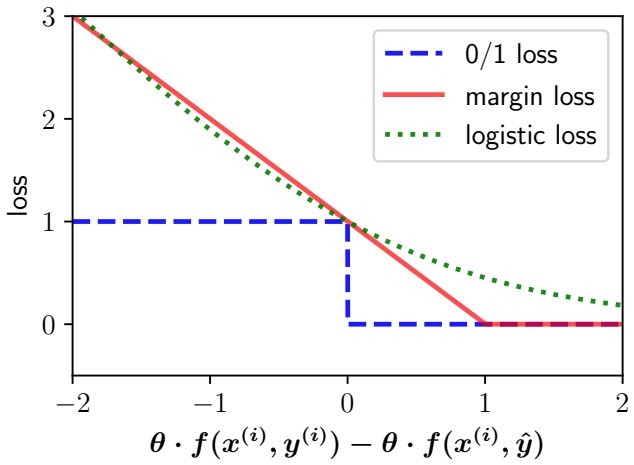

Figure 2.2: Margin, zero-one, and logistic loss functions.

into a loss function,

$$\ell_{\text{MARGIN}}(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) = \begin{cases} 0, & \gamma(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) \geq 1, \\ 1 - \gamma(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}), & \text{otherwise} \end{cases} \quad [2.44]$$

$$= \Big(1 - \gamma(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)})\Big)_+, \quad [2.45]$$

where $(x)_+ = \max(0, x)$. The loss is zero if there is a margin of at least 1 between the score for the true label and the best-scoring alternative $\hat{y}$. This is almost identical to the perceptron loss, but the hinge point is shifted to the right, as shown in Figure 2.2. The margin loss is a convex upper bound on the zero-one loss.

The margin loss can be minimized using an online learning rule that is similar to perceptron. We will call this learning rule the **online support vector machine**, for reasons that will be discussed in the derivation. Let us first generalize the notion of a classification error with a **cost function** $c(y^{(i)}, y)$. We will focus on the simple cost function,

$$c(y^{(i)}, y) = \begin{cases} 1, & y^{(i)} \neq \hat{y} \\ 0, & \text{otherwise}, \end{cases} \quad [2.46]$$

but it is possible to design specialized cost functions that assign heavier penalties to especially undesirable errors (Tsochantaridis et al., 2004). This idea is revisited in chapter 7.

Using the cost function, we can now define the online support vector machine as the

following classification rule:

$$\hat{y} = \underset{y \in \mathcal{Y}}{\text{argmax}}\, \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y) + c(y^{(i)}, y) \quad [2.47]$$

$$\boldsymbol{\theta}^{(t)} \leftarrow (1-\lambda)\boldsymbol{\theta}^{(t-1)} + \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \boldsymbol{f}(\boldsymbol{x}^{(i)}, \hat{y}) \quad [2.48]$$

This update is similar in form to the perceptron, with two key differences.

- Rather than selecting the label $\hat{y}$ that maximizes the score of the current classification model, the argmax searches for labels that are both *strong*, as measured by $\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y)$, and *wrong*, as measured by $c(y^{(i)}, y)$. This maximization is known as **cost-augmented decoding**, because it augments the maximization objective to favor high-cost labels. If the highest-scoring label is $y = y^{(i)}$, then the margin loss for this instance is zero, and no update is needed. If not, then an update is required to reduce the margin loss — even if the current model classifies the instance correctly. Cost augmentation is only done while learning; it is not applied when making predictions on unseen data.
- The previous weights $\boldsymbol{\theta}^{(t-1)}$ are scaled by $(1-\lambda)$, with $\lambda \in (0, 1)$. The effect of this term is to cause the weights to "decay" back towards zero. In the support vector machine, this term arises from the minimization of a specific form of the margin, as described below. However, it can also be viewed as a form of **regularization**, which can help to prevent overfitting (see § 2.5.1). In this sense, it plays a role that is similar to smoothing in Naïve Bayes (see § 2.2.4).

2.4.2 *Derivation of the online support vector machine

The derivation of the online support vector machine is somewhat involved, but gives further intuition about why the method works. Begin by returning the idea of linear separability (Definition 1): if a dataset is linearly separable, then there is some hyperplane $\boldsymbol{\theta}$ that correctly classifies all training instances with margin ρ. This margin can be increased to any desired value by multiplying the weights by a constant.

Now, for any datapoint $(\boldsymbol{x}^{(i)}, y^{(i)})$, the geometric distance to the separating hyperplane is given by $\frac{\gamma(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)})}{||\boldsymbol{\theta}||_2}$, where the denominator is the norm of the weights, $||\boldsymbol{\theta}||_2 = \sqrt{\sum_j \theta_j^2}$. The geometric distance is sometimes called the **geometric margin**, in contrast to the **functional margin** $\gamma(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)})$. Both are shown in Figure 2.3. The geometric margin is a good measure of the robustness of the separator: if the functional margin is large, but the norm $||\boldsymbol{\theta}||_2$ is also large, then a small change in $\boldsymbol{x}^{(i)}$ could cause it to be misclassified. We therefore seek to maximize the minimum geometric margin across the dataset, subject

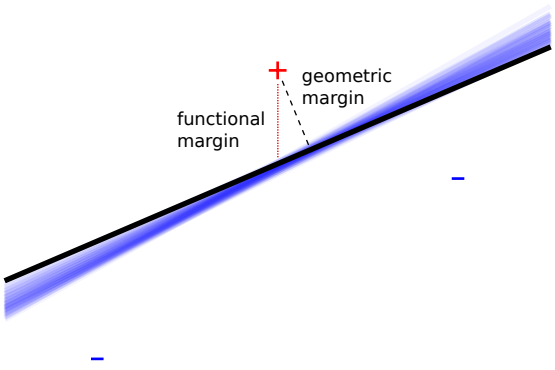

Figure 2.3: Functional and geometric margins for a binary classification problem. All separators that satisfy the margin constraint are shown. The separator with the largest geometric margin is shown in bold.

to the constraint that the margin loss is always zero:

$$\begin{aligned} \max_{\boldsymbol{\theta}} \quad & \min_{i=1,2,\ldots N} \quad \frac{\gamma(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)})}{||\boldsymbol{\theta}||_2} \\ \text{s.t.} \quad & \gamma(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) \geq 1, \quad \forall i. \end{aligned} \tag{2.49}$$

This is a **constrained optimization** problem, where the second line describes constraints on the space of possible solutions $\boldsymbol{\theta}$. In this case, the constraint is that the functional margin always be at least one, and the objective is that the minimum geometric margin be as large as possible.

Constrained optimization is reviewed in Appendix B. In this case, further manipulation yields an unconstrained optimization problem. First, note that the norm $||\boldsymbol{\theta}||_2$ scales linearly: $||a\boldsymbol{\theta}||_2 = a||\boldsymbol{\theta}||_2$. Furthermore, the functional margin γ is a linear function of $\boldsymbol{\theta}$, so that $\gamma(a\boldsymbol{\theta}, \boldsymbol{x}^{(i)}, y^{(i)}) = a\gamma(\boldsymbol{\theta}, \boldsymbol{x}^{(i)}, y^{(i)})$. As a result, any scaling factor on $\boldsymbol{\theta}$ will cancel in the numerator and denominator of the geometric margin. If the data is linearly separable at any $\rho > 0$, it is always possible to rescale the functional margin to 1 by multiplying $\boldsymbol{\theta}$ by a scalar constant. We therefore need only minimize the denominator $||\boldsymbol{\theta}||_2$, subject to the constraint on the functional margin. The minimizer of $||\boldsymbol{\theta}||_2$ is also the minimizer of $\frac{1}{2}||\boldsymbol{\theta}||_2^2 = \frac{1}{2}\sum \theta_j^2$, which is easier to work with. This yields a simpler optimization prob-

lem:

$$\min_{\boldsymbol{\theta}} . \quad \frac{1}{2}||\boldsymbol{\theta}||_2^2$$
$$\text{s.t.} \quad \gamma(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) \geq 1, \quad \forall_i. \tag{2.50}$$

This problem is a **quadratic program**: the objective is a quadratic function of the parameters, and the constraints are all linear inequalities. One solution to this problem is to incorporate the constraints through Lagrange multipliers $\alpha_i \geq 0, i = 1, 2, \ldots, N$. The instances for which $\alpha_i > 0$ are called **support vectors**; other instances are irrelevant to the classification boundary. This motivates the name **support vector machine**.

Thus far we have assumed linear separability, but many datasets of interest are not linearly separable. In this case, there is no $\boldsymbol{\theta}$ that satisfies the margin constraint. To add more flexibility, we can introduce a set of **slack variables** $\xi_i \geq 0$. Instead of requiring that the functional margin be greater than or equal to one, we require that it be greater than or equal to $1 - \xi_i$. Ideally there would not be any slack, so the slack variables are penalized in the objective function:

$$\min_{\boldsymbol{\theta}, \boldsymbol{\xi}} \quad \frac{1}{2}||\boldsymbol{\theta}||_2^2 + C \sum_{i=1}^{N} \xi_i$$
$$\text{s.t.} \quad \gamma(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) + \xi_i \geq 1, \quad \forall i$$
$$\xi_i \geq 0, \quad \forall_i. \tag{2.51}$$

The hyperparameter C controls the tradeoff between violations of the margin constraint and the preference for a low norm of $\boldsymbol{\theta}$. As $C \to \infty$, slack is infinitely expensive, and there is only a solution if the data is separable. As $C \to 0$, slack becomes free, and there is a trivial solution at $\boldsymbol{\theta} = \mathbf{0}$. Thus, C plays a similar role to the smoothing parameter in Naïve Bayes (§ 2.2.4), trading off between a close fit to the training data and better generalization. Like the smoothing parameter of Naïve Bayes, C must be set by the user, typically by maximizing performance on a heldout development set.

To solve the constrained optimization problem defined in Equation 2.51, we can first solve for the slack variables,

$$\xi_i \geq (1 - \gamma(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}))_+. \tag{2.52}$$

The inequality is tight: the optimal solution is to make the slack variables as small as possible, while still satisfying the constraints (Ratliff et al., 2007; Smith, 2011). By plugging in the minimum slack variables back into Equation 2.51, the problem can be transformed into the unconstrained optimization,

$$\min_{\boldsymbol{\theta}} \quad \frac{\lambda}{2}||\boldsymbol{\theta}||_2^2 + \sum_{i=1}^{N} (1 - \gamma(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}))_+, \tag{2.53}$$

where each ξ_i has been substituted by the right-hand side of Equation 2.52, and the factor of C on the slack variables has been replaced by an equivalent factor of $\lambda = \frac{1}{C}$ on the norm of the weights.

Equation 2.53 can be rewritten by expanding the margin,

$$\min_{\boldsymbol{\theta}} \quad \frac{\lambda}{2}||\boldsymbol{\theta}||_2^2 + \sum_{i=1}^{N} \left(\max_{y \in \mathcal{Y}} \left(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y) + c(y^{(i)}, y) \right) - \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) \right)_+, \tag{2.54}$$

where $c(y, y^{(i)})$ is the cost function defined in Equation 2.46. We can now differentiate with respect to the weights,

$$\nabla_{\boldsymbol{\theta}} L_{\text{SVM}} = \lambda \boldsymbol{\theta} + \sum_{i=1}^{N} \boldsymbol{f}(\boldsymbol{x}^{(i)}, \hat{y}) - \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}), \tag{2.55}$$

where L_{SVM} refers to minimization objective in Equation 2.54 and $\hat{y} = \text{argmax}_{y \in \mathcal{Y}} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y) + c(y^{(i)}, y)$. The online support vector machine update arises from the application of **stochastic gradient descent** (described in § 2.6.2) to this gradient.

2.5 Logistic regression

Thus far, we have seen two broad classes of learning algorithms. Naïve Bayes is a probabilistic method, where learning is equivalent to estimating a joint probability distribution. The perceptron and support vector machine are discriminative, error-driven algorithms: the learning objective is closely related to the number of errors on the training data. Probabilistic and error-driven approaches each have advantages: probability makes it possible to quantify uncertainty about the predicted labels, but the probability model of Naïve Bayes makes unrealistic independence assumptions that limit the features that can be used.

Logistic regression combines advantages of discriminative and probabilistic classifiers. Unlike Naïve Bayes, which starts from the **joint probability** $\text{p}_{X,Y}$, logistic regression defines the desired **conditional probability** $\text{p}_{Y|X}$ directly. Think of $\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y)$ as a scoring function for the compatibility of the base features $\boldsymbol{x}$ and the label y. To convert this score into a probability, we first exponentiate, obtaining $\exp(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y))$, which is guaranteed to be non-negative. Next, we normalize, dividing over all possible labels $y' \in \mathcal{Y}$. The resulting conditional probability is defined as,

$$\text{p}(y \mid \boldsymbol{x}; \boldsymbol{\theta}) = \frac{\exp(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y))}{\sum_{y' \in \mathcal{Y}} \exp(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y'))}. \tag{2.56}$$

Given a dataset $\mathcal{D} = \{(\boldsymbol{x}^{(i)}, y^{(i)})\}_{i=1}^{N}$, the weights $\boldsymbol{\theta}$ are estimated by **maximum conditional likelihood**,

$$\log \mathrm{p}(\boldsymbol{y}^{(1:N)} \mid \boldsymbol{x}^{(1:N)}; \boldsymbol{\theta}) = \sum_{i=1}^{N} \log \mathrm{p}(y^{(i)} \mid \boldsymbol{x}^{(i)}; \boldsymbol{\theta}) \quad [2.57]$$

$$= \sum_{i=1}^{N} \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \log \sum_{y' \in \mathcal{Y}} \exp\left(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y')\right). \quad [2.58]$$

The final line is obtained by plugging in Equation 2.56 and taking the logarithm.[16] Inside the sum, we have the (additive inverse of the) **logistic loss**,

$$\ell_{\text{LOGREG}}(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) = -\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) + \log \sum_{y' \in \mathcal{Y}} \exp(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y')) \quad [2.59]$$

The logistic loss is shown in Figure 2.2 on page 31. A key difference from the zero-one and hinge losses is that logistic loss is never zero. This means that the objective function can always be improved by assigning higher confidence to the correct label.

2.5.1 Regularization

As with the support vector machine, better generalization can be obtained by penalizing the norm of $\boldsymbol{\theta}$. This is done by adding a multiple of the squared norm $\frac{\lambda}{2}||\boldsymbol{\theta}||_2^2$ to the minimization objective. This is called L_2 regularization, because $||\boldsymbol{\theta}||_2^2$ is the squared L_2 norm of the vector $\boldsymbol{\theta}$. Regularization forces the estimator to trade off performance on the training data against the norm of the weights, and this can help to prevent overfitting. Consider what would happen to the unregularized weight for a base feature j that is active in only one instance $\boldsymbol{x}^{(i)}$: the conditional log-likelihood could always be improved by increasing the weight for this feature, so that $\boldsymbol{\theta}_{(j,y^{(i)})} \rightarrow \infty$ and $\boldsymbol{\theta}_{(j,\tilde{y} \neq y^{(i)})} \rightarrow -\infty$, where (j, y) is the index of feature associated with $x_j^{(i)}$ and label y in $\boldsymbol{f}(\boldsymbol{x}^{(i)}, y)$.

In § 2.2.4 (footnote 11), we saw that smoothing the probabilities of a Naïve Bayes classifier can be justified as a form of maximum a posteriori estimation, in which the parameters of the classifier are themselves random variables, drawn from a **prior distribution**. The same justification applies to L_2 regularization. In this case, the prior is a zero-mean Gaussian on each term of $\boldsymbol{\theta}$. The log-likelihood under a zero-mean Gaussian is,

$$\log N(\theta_j; 0, \sigma^2) \propto -\frac{1}{2\sigma^2}\theta_j^2, \quad [2.60]$$

so that the regularization weight λ is equal to the inverse variance of the prior, $\lambda = \frac{1}{\sigma^2}$.

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

2.5.2 Gradients

Logistic loss is minimized by optimization along the gradient. Specific algorithms are described in the next section, but first let's compute the gradient with respect to the logistic loss of a single example:

$$\ell_{\text{LOGREG}} = -\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) + \log \sum_{y' \in \mathcal{Y}} \exp\Big(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y')\Big) \tag{2.61}$$

$$\frac{\partial \ell}{\partial \boldsymbol{\theta}} = -\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) + \frac{1}{\sum_{y'' \in \mathcal{Y}} \exp\left(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y'')\right)} \times \sum_{y' \in \mathcal{Y}} \exp\Big(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y')\Big) \times \boldsymbol{f}(\boldsymbol{x}^{(i)}, y') \tag{2.62}$$

$$= -\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) + \sum_{y' \in \mathcal{Y}} \frac{\exp\left(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y')\right)}{\sum_{y'' \in \mathcal{Y}} \exp\left(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y'')\right)} \times \boldsymbol{f}(\boldsymbol{x}^{(i)}, y') \tag{2.63}$$

$$= -\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) + \sum_{y' \in \mathcal{Y}} \text{p}(y' \mid \boldsymbol{x}^{(i)}; \boldsymbol{\theta}) \times \boldsymbol{f}(\boldsymbol{x}^{(i)}, y') \tag{2.64}$$

$$= -\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) + E_{Y|X}[\boldsymbol{f}(\boldsymbol{x}^{(i)}, y)]. \tag{2.65}$$

The final step employs the definition of a conditional expectation (§ A.5). The gradient of the logistic loss is equal to the difference between the expected counts under the current model, $E_{Y|X}[\boldsymbol{f}(\boldsymbol{x}^{(i)}, y)]$, and the observed feature counts $\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)})$. When these two vectors are equal for a single instance, there is nothing more to learn from it; when they are equal in sum over the entire dataset, there is nothing more to learn from the dataset as a whole. The gradient of the hinge loss is nearly identical, but it involves the features of the predicted label under the current model, $\boldsymbol{f}(\boldsymbol{x}^{(i)}, \hat{y})$, rather than the expected features $E_{Y|X}[\boldsymbol{f}(\boldsymbol{x}^{(i)}, y)]$ under the conditional distribution $\text{p}(y \mid \boldsymbol{x}; \boldsymbol{\theta})$.

The regularizer contributes $\lambda\boldsymbol{\theta}$ to the overall gradient:

$$L_{\text{LOGREG}} = \frac{\lambda}{2}||\boldsymbol{\theta}||_2^2 - \sum_{i=1}^{N} \left(\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \log \sum_{y' \in \mathcal{Y}} \exp \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}^{(i)}, y')\right) \tag{2.66}$$

$$\nabla_{\boldsymbol{\theta}} L_{\text{LOGREG}} = \lambda\boldsymbol{\theta} - \sum_{i=1}^{N} \Big(\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - E_{y|\boldsymbol{x}}[\boldsymbol{f}(\boldsymbol{x}^{(i)}, y)]\Big). \tag{2.67}$$

2.6 Optimization

Each of the classification algorithms in this chapter can be viewed as an optimization problem:

- In Naïve Bayes, the objective is the joint likelihood $\log \text{p}(\boldsymbol{x}^{(1:N)}, \boldsymbol{y}^{(1:N)})$. Maximum likelihood estimation yields a closed-form solution for $\boldsymbol{\theta}$.

- In the support vector machine, the objective is the regularized margin loss,

$$L_{\text{SVM}} = \frac{\lambda}{2}||\boldsymbol{\theta}||_2^2 + \sum_{i=1}^{N}(\max_{y\in\mathcal{Y}}(\boldsymbol{\theta}\cdot\boldsymbol{f}(\boldsymbol{x}^{(i)},y) + c(y^{(i)},y)) - \boldsymbol{\theta}\cdot\boldsymbol{f}(\boldsymbol{x}^{(i)},y^{(i)}))_+, \quad [2.68]$$

  There is no closed-form solution, but the objective is convex. The perceptron algorithm minimizes a similar objective.

- In logistic regression, the objective is the regularized negative log-likelihood,

$$L_{\text{LOGREG}} = \frac{\lambda}{2}||\boldsymbol{\theta}||_2^2 - \sum_{i=1}^{N}\left(\boldsymbol{\theta}\cdot\boldsymbol{f}(\boldsymbol{x}^{(i)},y^{(i)}) - \log\sum_{y\in\mathcal{Y}}\exp\left(\boldsymbol{\theta}\cdot\boldsymbol{f}(\boldsymbol{x}^{(i)},y)\right)\right) \quad [2.69]$$

  Again, there is no closed-form solution, but the objective is convex.

These learning algorithms are distinguished by *what* is being optimized, rather than *how* the optimal weights are found. This decomposition is an essential feature of contemporary machine learning. The domain expert's job is to design an objective function — or more generally, a **model** of the problem. If the model has certain characteristics, then generic optimization algorithms can be used to find the solution. In particular, if an objective function is differentiable, then gradient-based optimization can be employed; if it is also convex, then gradient-based optimization is guaranteed to find the globally optimal solution. The support vector machine and logistic regression have both of these properties, and so are amenable to generic **convex optimization** techniques (Boyd and Vandenberghe, 2004).

2.6.1 Batch optimization

In **batch optimization**, each update to the weights is based on a computation involving the entire dataset. One such algorithm is **gradient descent**, which iteratively updates the weights,

$$\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} - \eta^{(t)}\nabla_{\boldsymbol{\theta}}L, \quad [2.70]$$

where $\nabla_{\boldsymbol{\theta}}L$ is the gradient computed over the entire training set, and $\eta^{(t)}$ is the **learning rate** at iteration t. If the objective L is a convex function of $\boldsymbol{\theta}$, then this procedure is guaranteed to terminate at the global optimum, for appropriate schedule of learning rates, $\eta^{(t)}$.[17]

[17] Convergence proofs typically require the learning rate to satisfy the following conditions: $\sum_{t=1}^{\infty}\eta^{(t)} = \infty$ and $\sum_{t=1}^{\infty}(\eta^{(t)})^2 < \infty$ (Bottou et al., 2016). These properties are satisfied by any learning rate schedule $\eta^{(t)} = \eta^{(0)}t^{-\alpha}$ for $\alpha \in [1, 2]$.

In practice, gradient descent can be slow to converge, as the gradient can become infinitesimally small. Faster convergence can be obtained by second-order Newton optimization, which incorporates the inverse of the **Hessian matrix**,

$$H_{i,j} = \frac{\partial^2 L}{\partial \theta_i \partial \theta_j} \quad [2.71]$$

The size of the Hessian matrix is quadratic in the number of features. In the bag-of-words representation, this is usually too big to store, let alone invert. **Quasi-Network optimization** techniques maintain a low-rank approximation to the inverse of the Hessian matrix. Such techniques usually converge more quickly than gradient descent, while remaining computationally tractable even for large feature sets. A popular quasi-Newton algorithm is L-BFGS (Liu and Nocedal, 1989), which is implemented in many scientific computing environments, such as SCIPY and MATLAB.

For any gradient-based technique, the user must set the learning rates $\eta^{(t)}$. While convergence proofs usually employ a decreasing learning rate, in practice, it is common to fix $\eta^{(t)}$ to a small constant, like 10^{-3}. The specific constant can be chosen by experimentation, although there is research on determining the learning rate automatically (Schaul et al., 2013; Wu et al., 2018).

2.6.2 Online optimization

Batch optimization computes the objective on the entire training set before making an update. This may be inefficient, because at early stages of training, a small number of training examples could point the learner in the correct direction. **Online learning** algorithms make updates to the weights while iterating through the training data. The theoretical basis for this approach is a stochastic approximation to the true objective function,

$$\sum_{i=1}^{N} \ell(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) \approx N \times \ell(\boldsymbol{\theta}; \boldsymbol{x}^{(j)}, y^{(j)}), \qquad (\boldsymbol{x}^{(j)}, y^{(j)}) \sim \{(\boldsymbol{x}^{(i)}, y^{(i)})\}_{i=1}^{N}, \quad [2.72]$$

where the instance $(\boldsymbol{x}^{(j)}, y^{(j)})$ is sampled at random from the full dataset.

In **stochastic gradient descent**, the approximate gradient is computed by randomly sampling a single instance, and an update is made immediately. This is similar to the perceptron algorithm, which also updates the weights one instance at a time. In **minibatch** stochastic gradient descent, the gradient is computed over a small set of instances. A typical approach is to set the minibatch size so that the entire batch fits in memory on a graphics processing unit (GPU; Neubig et al., 2017). It is then possible to speed up learning by parallelizing the computation of the gradient over each instance in the minibatch.

Algorithm 5 offers a generalized view of gradient descent. In standard gradient descent, the batcher returns a single batch with all the instances. In stochastic gradient de-

Algorithm 5 Generalized gradient descent. The function BATCHER partitions the training set into B batches such that each instance appears in exactly one batch. In gradient descent, $B = 1$; in stochastic gradient descent, $B = N$; in minibatch stochastic gradient descent, $1 < B < N$.

procedure GRADIENT-DESCENT($\boldsymbol{x}^{(1:N)}, \boldsymbol{y}^{(1:N)}, L, \eta^{(1...\infty)}$, BATCHER, T_{max})
  $\boldsymbol{\theta} \leftarrow \mathbf{0}$
  $t \leftarrow 0$
  repeat
    $(\boldsymbol{b}^{(1)}, \boldsymbol{b}^{(2)}, \ldots, \boldsymbol{b}^{(B)}) \leftarrow$ BATCHER(N)
    for $n \in \{1, 2, \ldots, B\}$ **do**
      $t \leftarrow t + 1$
      $\boldsymbol{\theta}^{(t)} \leftarrow \boldsymbol{\theta}^{(t-1)} - \eta^{(t)} \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^{(t-1)}; \boldsymbol{x}^{(b_1^{(n)}, b_2^{(n)}, \ldots)}, \boldsymbol{y}^{(b_1^{(n)}, b_2^{(n)}, \ldots)})$
      if Converged$(\boldsymbol{\theta}^{(1,2,\ldots,t)})$ **then**
        return $\boldsymbol{\theta}^{(t)}$
  until $t \geq T_{\text{max}}$
  return $\boldsymbol{\theta}^{(t)}$

scent, it returns N batches with one instance each. In mini-batch settings, the batcher returns B minibatches, $1 < B < N$.

There are many other techniques for online learning, and research in this area is ongoing (Bottou et al., 2016). Some algorithms use an adaptive learning rate, which can be different for every feature (Duchi et al., 2011). Features that occur frequently are likely to be updated frequently, so it is best to use a small learning rate; rare features will be updated infrequently, so it is better to take larger steps. The **AdaGrad** (adaptive gradient) algorithm achieves this behavior by storing the sum of the squares of the gradients for each feature, and rescaling the learning rate by its inverse:

$$\boldsymbol{g}_t = \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^{(t)}; \boldsymbol{x}^{(i)}, y^{(i)}) \tag{2.73}$$

$$\theta_j^{(t+1)} \leftarrow \theta_j^{(t)} - \frac{\eta^{(t)}}{\sqrt{\sum_{t'=1}^{t} g_{t,j}^2}} g_{t,j}, \tag{2.74}$$

where j iterates over features in $\boldsymbol{f}(\boldsymbol{x}, y)$.

In most cases, the number of active features for any instance is much smaller than the number of weights. If so, the computation cost of online optimization will be dominated by the update from the regularization term, $\lambda\boldsymbol{\theta}$. The solution is to be "lazy", updating each θ_j only as it is used. To implement lazy updating, store an additional parameter τ_j, which is the iteration at which θ_j was last updated. If θ_j is needed at time t, the $t - \tau$ regularization updates can be performed all at once. This strategy is described in detail by Kummerfeld et al. (2015).

2.7 *Additional topics in classification

This section presents some additional topics in classification that are particularly relevant for natural language processing, especially for understanding the research literature.

2.7.1 Feature selection by regularization

In logistic regression and large-margin classification, generalization can be improved by regularizing the weights towards 0, using the L_2 norm. But rather than encouraging weights to be small, it might be better for the model to be **sparse**: it should assign weights of exactly zero to most features, and only assign non-zero weights to features that are clearly necessary. This idea can be formalized by the L_0 norm, $L_0 = ||\boldsymbol{\theta}||_0 = \sum_j \delta(\theta_j \neq 0)$, which applies a constant penalty for each non-zero weight. This norm can be thought of as a form of **feature selection**: optimizing the L_0-regularized conditional likelihood is equivalent to trading off the log-likelihood against the number of active features. Reducing the number of active features is desirable because the resulting model will be fast, low-memory, and should generalize well, since irrelevant features will be pruned away. Unfortunately, the L_0 norm is non-convex and non-differentiable. Optimization under L_0 regularization is **NP-hard**, meaning that it can be solved efficiently only if P=NP (Ge et al., 2011).

A useful alternative is the L_1 norm, which is equal to the sum of the absolute values of the weights, $||\boldsymbol{\theta}||_1 = \sum_j |\theta_j|$. The L_1 norm is convex, and can be used as an approximation to L_0 (Tibshirani, 1996). Conveniently, the L_1 norm also performs feature selection, by driving many of the coefficients to zero; it is therefore known as a **sparsity inducing regularizer**. The L_1 norm does not have a gradient at $\theta_j = 0$, so we must instead optimize the L_1-regularized objective using **subgradient** methods. The associated stochastic subgradient descent algorithms are only somewhat more complex than conventional SGD; Sra et al. (2012) survey approaches for estimation under L_1 and other regularizers.

Gao et al. (2007) compare L_1 and L_2 regularization on a suite of NLP problems, finding that L_1 regularization generally gives similar accuracy to L_2 regularization, but that L_1 regularization produces models that are between ten and fifty times smaller, because more than 90% of the feature weights are set to zero.

2.7.2 Other views of logistic regression

In binary classification, we can dispense with the feature function, and choose y based on the inner product of $\boldsymbol{\theta} \cdot \boldsymbol{x}$. The conditional probability $\mathrm{p}_{Y|X}$ is obtained by passing this

inner product through a **logistic function**,

$$\sigma(a) \triangleq \frac{\exp(a)}{1+\exp(a)} = (1+\exp(-a))^{-1} \quad [2.75]$$

$$\mathrm{p}(y \mid \boldsymbol{x}; \boldsymbol{\theta}) = \sigma(\boldsymbol{\theta} \cdot \boldsymbol{x}). \quad [2.76]$$

This is the origin of the name "logistic regression." Logistic regression can be viewed as part of a larger family of **generalized linear models** (GLMs), in which various other **link functions** convert between the inner product $\boldsymbol{\theta} \cdot \boldsymbol{x}$ and the parameter of a conditional probability distribution.

Logistic regression and related models are sometimes referred to as **log-linear**, because the log-probability is a linear function of the features. But in the early NLP literature, logistic regression was often called **maximum entropy** classification (Berger et al., 1996). This name refers to an alternative formulation, in which the goal is to find the maximum entropy probability function that satisfies **moment-matching** constraints. These constraints specify that the empirical counts of each feature should match the expected counts under the induced probability distribution $\mathrm{p}_{Y|X;\boldsymbol{\theta}}$,

$$\sum_{i=1}^{N} f_j(\boldsymbol{x}^{(i)}, y^{(i)}) = \sum_{i=1}^{N} \sum_{y \in \mathcal{Y}} \mathrm{p}(y \mid \boldsymbol{x}^{(i)}; \boldsymbol{\theta}) f_j(\boldsymbol{x}^{(i)}, y), \quad \forall j \quad [2.77]$$

The moment-matching constraint is satisfied exactly when the derivative of the conditional log-likelihood function (Equation 2.65) is equal to zero. However, the constraint can be met by many values of $\boldsymbol{\theta}$, so which should we choose?

The **entropy** of the conditional probability distribution $\mathrm{p}_{Y|X}$ is,

$$H(\mathrm{p}_{Y|X}) = -\sum_{\boldsymbol{x} \in \mathcal{X}} \mathrm{p}_X(\boldsymbol{x}) \sum_{y \in \mathcal{Y}} \mathrm{p}_{Y|X}(y \mid \boldsymbol{x}) \log \mathrm{p}_{Y|X}(y \mid \boldsymbol{x}), \quad [2.78]$$

where $\mathcal{X}$ is the set of all possible feature vectors, and $\mathrm{p}_X(\boldsymbol{x})$ is the probability of observing the base features $\boldsymbol{x}$. The distribution p_X is unknown, but it can be estimated by summing over all the instances in the training set,

$$\tilde{H}(\mathrm{p}_{Y|X}) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{y \in \mathcal{Y}} \mathrm{p}_{Y|X}(y \mid \boldsymbol{x}^{(i)}) \log \mathrm{p}_{Y|X}(y \mid \boldsymbol{x}^{(i)}). \quad [2.79]$$

If the entropy is large, the likelihood function is smooth across possible values of y; if it is small, the likelihood function is sharply peaked at some preferred value; in the limiting case, the entropy is zero if $\mathrm{p}(y \mid x) = 1$ for some y. The maximum-entropy criterion chooses to make the weakest commitments possible, while satisfying the moment-matching constraints from Equation 2.77. The solution to this constrained optimization problem is identical to the maximum conditional likelihood (logistic-loss) formulation that was presented in § 2.5.

2.8 Summary of learning algorithms

It is natural to ask which learning algorithm is best, but the answer depends on what characteristics are important to the problem you are trying to solve.

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

Perceptron *Pros*: easy to implement; online; error-driven learning means that accuracy is typically high, especially after averaging. *Cons*: not probabilistic; hard to know when to stop learning; lack of margin can lead to overfitting.

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

Logistic regression *Pros*: error-driven and probabilistic; overfitting is controlled by a regularization parameter. *Cons*: batch learning requires black-box optimization; logistic loss can "overtrain" on correctly labeled examples.

One of the main distinctions is whether the learning algorithm offers a probability over labels. This is useful in modular architectures, where the output of one classifier is the input for some other system. In cases where probability is not necessary, the support vector machine is usually the right choice, since it is no more difficult to implement than the perceptron, and is often more accurate. When probability is necessary, logistic regression is usually more accurate than Naïve Bayes.

Additional resources

A machine learning textbook will offer more classifiers and more details (e.g., Murphy, 2012), although the notation will differ slightly from what is typical in natural language processing. Probabilistic methods are surveyed by Hastie et al. (2009), and Mohri et al. (2012) emphasize theoretical considerations. Bottou et al. (2016) surveys the rapidly moving field of online learning, and Kummerfeld et al. (2015) empirically review several optimization algorithms for large-margin learning. The python toolkit SCIKIT-LEARN includes implementations of all of the algorithms described in this chapter (Pedregosa et al., 2011).

Appendix B describes an alternative large-margin classifier, called **passive-aggressive**. Passive-aggressive is an online learner that seeks to make the smallest update that satisfies the margin constraint at the current instance. It is closely related to MIRA, which was used widely in NLP in the 2000s (Crammer and Singer, 2003).

Exercises

There will be exercises at the end of each chapter. In this chapter, the exercises are mostly mathematical, matching the subject material. In other chapters, the exercises will emphasize linguistics or programming.

1. Let $\boldsymbol{x}$ be a bag-of-words vector such that $\sum_{j=1}^{V} x_j = 1$. Verify that the multinomial probability $\text{p}_{\text{mult}}(\boldsymbol{x}; \boldsymbol{\phi})$, as defined in Equation 2.12, is identical to the probability of the same document under a categorical distribution, $\text{p}_{\text{cat}}(\boldsymbol{w}; \boldsymbol{\phi})$.

2. Suppose you have a single feature x, with the following conditional distribution:

$$\text{p}(x \mid y) = \begin{cases} \alpha, & X = 0, Y = 0 \\ 1 - \alpha, & X = 1, Y = 0 \\ 1 - \beta, & X = 0, Y = 1 \\ \beta, & X = 1, Y = 1. \end{cases} \quad [2.80]$$

   Further suppose that the prior is uniform, $\Pr(Y = 0) = \Pr(Y = 1) = \frac{1}{2}$, and that both $\alpha > \frac{1}{2}$ and $\beta > \frac{1}{2}$. Given a Naïve Bayes classifier with accurate parameters, what is the probability of making an error?

3. Derive the maximum-likelihood estimate for the parameter $\boldsymbol{\mu}$ in Naïve Bayes.

4. The classification models in the text have a vector of weights for each possible label. While this is notationally convenient, it is overdetermined: for any linear classifier that can be obtained with $K \times V$ weights, an equivalent classifier can be constructed using $(K - 1) \times V$ weights.

   a) Describe how to construct this classifier. Specifically, if given a set of weights $\boldsymbol{\theta}$ and a feature function $\boldsymbol{f}(\boldsymbol{x}, y)$, explain how to construct alternative weights and feature function $\boldsymbol{\theta}'$ and $\boldsymbol{f}'(\boldsymbol{x}, y)$, such that,

$$\forall y, y' \in \mathcal{Y}, \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y) - \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y') = \boldsymbol{\theta}' \cdot \boldsymbol{f}'(\boldsymbol{x}, y) - \boldsymbol{\theta}' \cdot \boldsymbol{f}'(\boldsymbol{x}, y'). \quad [2.81]$$

   b) Explain how your construction justifies the well-known alternative form for binary logistic regression, $\Pr(Y = 1 \mid \boldsymbol{x}; \boldsymbol{\theta}) = \frac{1}{1+\exp(-\boldsymbol{\theta}' \cdot \boldsymbol{x})} = \sigma(\boldsymbol{\theta}' \cdot \boldsymbol{x})$, where σ is the sigmoid function.

5. Suppose you have two labeled datasets D_1 and D_2, with the same features and labels.

   - Let $\boldsymbol{\theta}^{(1)}$ be the unregularized logistic regression (LR) coefficients from training on dataset D_1.

- Let $\boldsymbol{\theta}^{(2)}$ be the unregularized LR coefficients (same model) from training on dataset D_2.
- Let $\boldsymbol{\theta}^*$ be the unregularized LR coefficients from training on the combined dataset $D_1 \cup D_2$.

Under these conditions, prove that for any feature j,

$$\theta_j^* \geq \min(\theta_j^{(1)}, \theta_j^{(2)})$$
$$\theta_j^* \leq \max(\theta_j^{(1)}, \theta_j^{(2)}).$$

6. Let $\hat{\boldsymbol{\theta}}$ be the solution to an unregularized logistic regression problem, and let $\boldsymbol{\theta}^*$ be the solution to the same problem, with L_2 regularization. Prove that $||\boldsymbol{\theta}^*||_2^2 \leq ||\hat{\boldsymbol{\theta}}||_2^2$.

7. As noted in the discussion of averaged perceptron in § 2.3.2, the computation of the running sum $\boldsymbol{m} \leftarrow \boldsymbol{m} + \boldsymbol{\theta}$ is unnecessarily expensive, requiring $K \times V$ operations. Give an alternative way to compute the averaged weights $\overline{\boldsymbol{\theta}}$, with complexity that is independent of V and linear in the sum of feature sizes $\sum_{i=1}^{N} |\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)})|$.

8. Consider a dataset that is comprised of two identical instances $\boldsymbol{x}^{(1)} = \boldsymbol{x}^{(2)}$ with distinct labels $y^{(1)} \neq y^{(2)}$. Assume all features are binary, $x_j \in \{0, 1\}$ for all j.

   Now suppose that the averaged perceptron always trains on the instance $(\boldsymbol{x}^{i(t)}, y^{i(t)})$, where $i(t) = 2 - (t \mod 2)$, which is 1 when the training iteration t is odd, and 2 when t is even. Further suppose that learning terminates under the following condition:

   $$\epsilon \geq \max_j \left| \frac{1}{t} \sum_t \theta_j^{(t)} - \frac{1}{t-1} \sum_t \theta_j^{(t-1)} \right|. \quad [2.82]$$

   In words, the algorithm stops when the largest change in the averaged weights is less than or equal to ϵ. Compute the number of iterations before the averaged perceptron terminates.

9. Prove that the margin loss is convex in $\boldsymbol{\theta}$. Use this definition of the margin loss:

   $$L(\boldsymbol{\theta}) = -\boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y^*) + \max_y \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y) + c(y^*, y), \quad [2.83]$$

   where y^* is the gold label. As a reminder, a function f is convex iff,

   $$f(\alpha x_1 + (1-\alpha)x_2) \leq \alpha f(x_1) + (1-\alpha) f(x_2), \quad [2.84]$$

   for any x_1, x_2 and $\alpha \in [0, 1]$.

10. If a function f is m-strongly convex, then for some $m > 0$, the following inequality holds for all x and x' on the domain of the function:

$$f(x') \leq f(x) + (\nabla_x f) \cdot (x' - x) + \frac{m}{2}||x' - x||_2^2. \quad [2.85]$$

Let $f(x) = L(\boldsymbol{\theta}^{(t)})$, representing the loss of the classifier at iteration t of gradient descent; let $f(x') = L(\boldsymbol{\theta}^{(t+1)})$. Assuming the loss function is m-convex, prove that $L(\boldsymbol{\theta}^{(t+1)}) \leq L(\boldsymbol{\theta}^{(t)})$ for an appropriate constant learning rate η, which will depend on m. Explain why this implies that gradient descent converges when applied to an m-strongly convex loss function with a unique minimum.