Machine Learning
the Fundamentals

William J. Deuschle

Harvard College
Cambridge, Massachusetts
April, 2019

Contents

3 Classification **1**

3.1 Defining the Problem . . . 1
3.2 Solution Options . . . 1
3.3 Discriminant Functions . . . 2
3.3.1 Basic Setup: Binary Linear Classification . . . 2
3.3.2 Multiple Classes . . . 3
3.3.3 Basis Changes in Classification . . . 3
3.4 Algorithms for Decision Boundaries . . . 6
3.4.1 Least Squares Loss . . . 6
3.4.2 Fisher's Linear Discriminant . . . 7
3.4.3 Perceptron Algorithm . . . 7
3.5 Probabilistic Methods . . . 9
3.5.1 Probabilistic Discriminative Models . . . 9
Logistic Regression . . . 10
Multi-Class Logistic Regression and Softmax . . . 12
3.5.2 Probabilistic Generative Models . . . 13
Classification in the Generative Setting . . . 13
Maximum Likelihood Solution . . . 14
Naive Bayes . . . 16
3.6 Conclusion . . . 18
3.7 Practice Problems . . . 18

Chapter 3

Classification

In the last chapter we explored ways of predicting a continuous, real-number target. In this chapter, we're going to think about a different problem- one where our target output is discrete-valued. This type of problem, one where we make a prediction by choosing between finite class options, is known as **classification**.

3.1 Defining the Problem

As we did when studying regression, let's begin by thinking about the type of problems we are trying to solve. Here are a few examples of classification tasks:

1. Predicting whether a given email is spam.

2. Predicting the type of object in an image.

3. Predicting whether a manufactured good is defective.

The point of classification is hopefully clear: we're trying to identify the most appropriate class for an input data point.

Definition 3.1.1 (Classification): A set of problems that seeks to make predictions about unobserved target classes given observed input variables.

3.2 Solution Options

There are several different means by which we can solve classification problems. We're going to discuss three in this chapter: discriminant functions, probabilistic discriminative models (also known as logistic regression), and probabilistic generative models. Note that these are not the only methods for performing classification tasks, but they are similar enough that it makes sense to present and explore them together. Specifically, these techniques all use some linear combination of input variables to produce a class prediction. For that reason, we will refer to these techniques as **generalized linear models**.

ML Framework Cube: Generalized Linear Models

Since we are using these techniques to perform classification, generalized linear models deal with a **discrete** output domain. Second, as with linear regression, our goal is to make predictions on future data points given an initial set of data to learn from. Thus, generalized linear models are **supervised** techniques. Finally, depending on the type of generalized linear model, they can be either **probabilistic or non-probabilistic**.

Domain	***Training***	***Probabilistic***
Discrete	Supervised	Yes / No

3.3 Discriminant Functions

Generalized linear models for classification come in several different flavors. The most straightforward method carries over very easily from linear regression: **discriminant functions**. As we will see, with discriminant functions we are linearly separating the input space into sections belonging to different target classes. We will explore this method first. One thing to keep in mind is that it's generally easiest to initially learn these techniques in the case where we have only two target classes, but there is typically a generalization that allows us to handle the multi-class case as well.

As with linear regression, discriminant functions $h(\mathbf{x}, \mathbf{w})$ seek to find a weighted combination of our input variables to make a prediction about the target class:

$$h(\mathbf{x}, \mathbf{w}) = w^{(0)}x^{(0)} + w^{(1)}x^{(1)} + ... + w^{(D)}x^{(D)} \tag{3.1}$$

where we are using the bias trick of appending $x^{(0)} = 1$ to all of our data points.

3.3.1 Basic Setup: Binary Linear Classification

The simplest use case for a discriminant function is when we only have two classes that we are trying to decide between. Let's denote these two classes **1** and **-1**. Our discriminant function in Equation 3.1 will then predict class **1** if $h(\mathbf{x}, \mathbf{w}) \geq 0$ and class **-1** if $h(\mathbf{x}, \mathbf{w}) < 0$:

$$\begin{cases} 1 & \text{if } h(\mathbf{x}, \mathbf{w}) \geq 0 \\ -1 & \text{if } h(\mathbf{x}, \mathbf{w}) < 0 \end{cases}$$

Geometrically, the linear separation between these two classes then looks like that of Figure 3.1. Notice the line where our prediction switches from class 1 to class -1. This is precisely where $h(\mathbf{x}, \mathbf{w}) = 0$, and it is known as the **decision boundary**.

Definition 3.3.1 (Decision Boundary): The decision boundary is the line that divides the input space into different target classes. It is learned from an initial data set, and then the target class of new data points can be predicted based on where they fall relative to the decision boundary. At the decision boundary, the discriminant function takes on a value of 0.

⋆ You will sometimes see the term **decision surface** in place of decision boundary, particularly if the input space is larger than two dimensions.

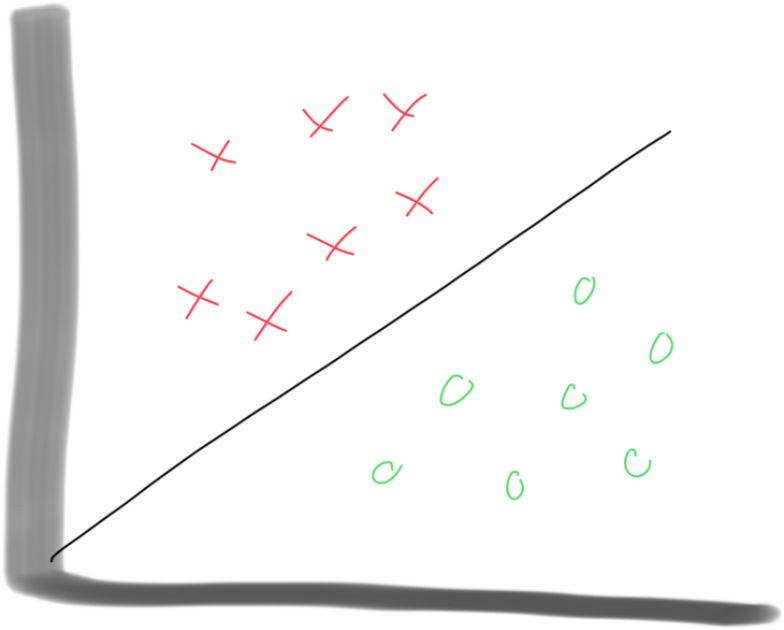

Figure 3.1: Clear separation between classes.

3.3.2 Multiple Classes

Now consider the case that we have $K > 2$ classes $C_1, C_2, ..., C_K$ to choose between. One obvious approach we might try is to use K different discriminant functions that simply determine whether or not a given input is in that class C_k. This is known as a *one-versus-all* approach, and it doesn't work properly because we end up with ambiguous regions as demonstrated in Figure 3.2.

Another obvious approach we might employ is to use $\binom{K}{2}$ discriminant functions that each determine whether a given point is more likely to be in class C_j or class C_k. This is known as a *one-versus-one* approach, and it also doesn't work because we again end up with ambiguous regions as demonstrated in Figure 3.3.

Instead, we can avoid these ambiguities in the multi-class case by using K different linear classifiers $h_k(\mathbf{x}, \mathbf{w}_k)$, and then assigning new data points to the class C_k for which $h_k(\mathbf{x}, \mathbf{w}_k) > h_j(\mathbf{x}, \mathbf{w}_j)$ for all $j \neq k$. Then, similar to the two-class case, the decision boundaries are described by the surface along which $h_k(\mathbf{x}, \mathbf{w}_k) = h_j(\mathbf{x}, \mathbf{w}_j)$.

Now that we've explored the multi-class generalization, we can discuss how to learn the weights w that define the optimal discriminant functions.

3.3.3 Basis Changes in Classification

We initially discussed basis changes in the context of linear regression, and they are equally important for classification tasks. For example, consider the data set in Figure 3.4.

It's obviously not possible for us to use a linear classifier to separate this data set. However, if we apply a basis change by squaring one of the data points, we instead have Figure 3.5, which is now linearly separable by a plane between the two classes. Applying a generic basis change $\phi(\cdot)$,

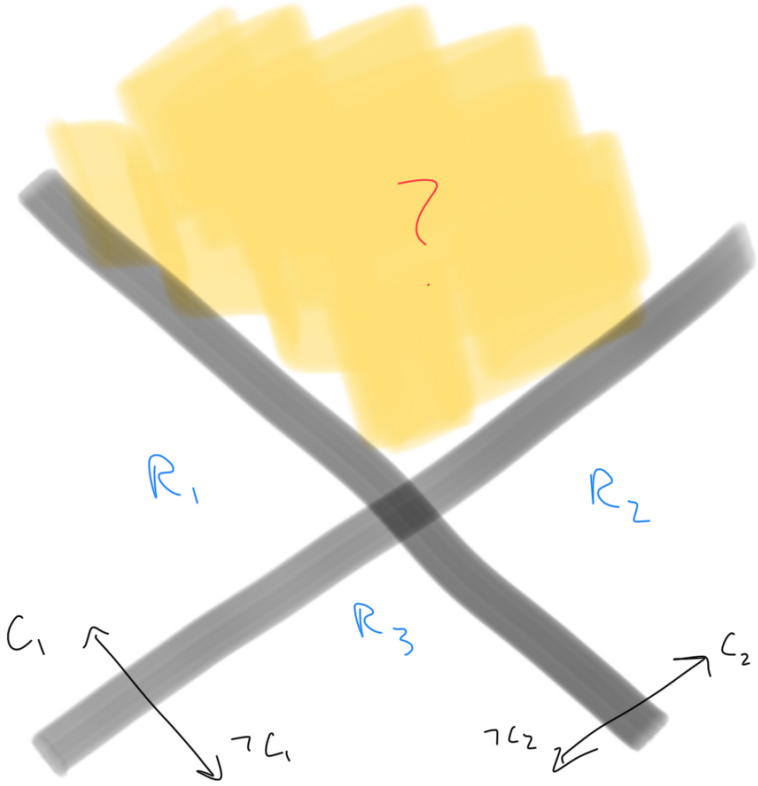

Figure 3.2: Ambiguities arise from one-versus-all method.

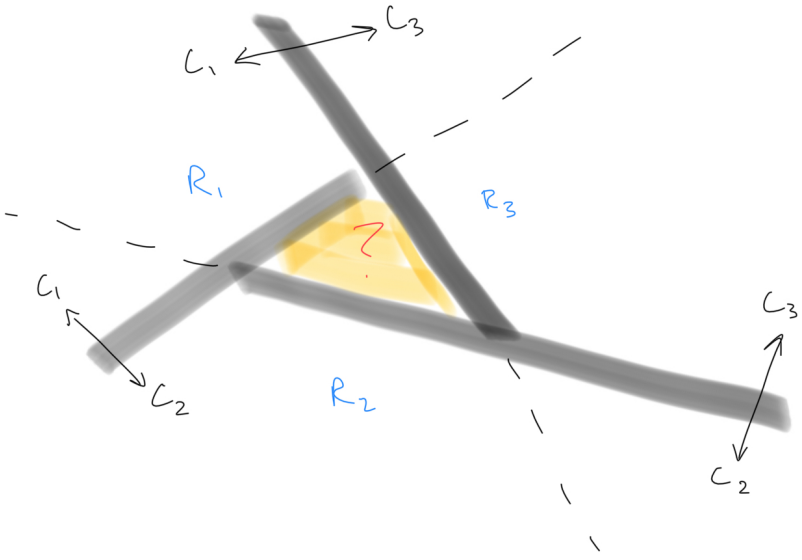

Figure 3.3: Ambiguities arise from one-versus-one method.

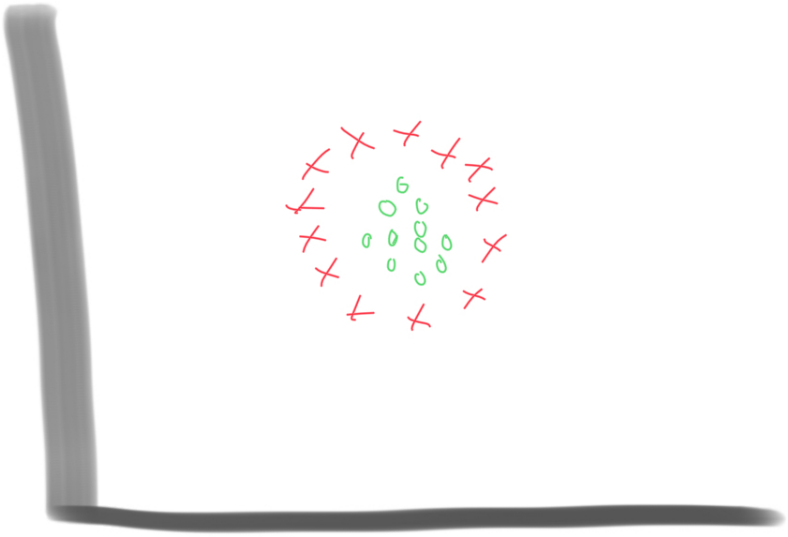

Figure 3.4: Data set without any basis functions applied, not linearly separable.

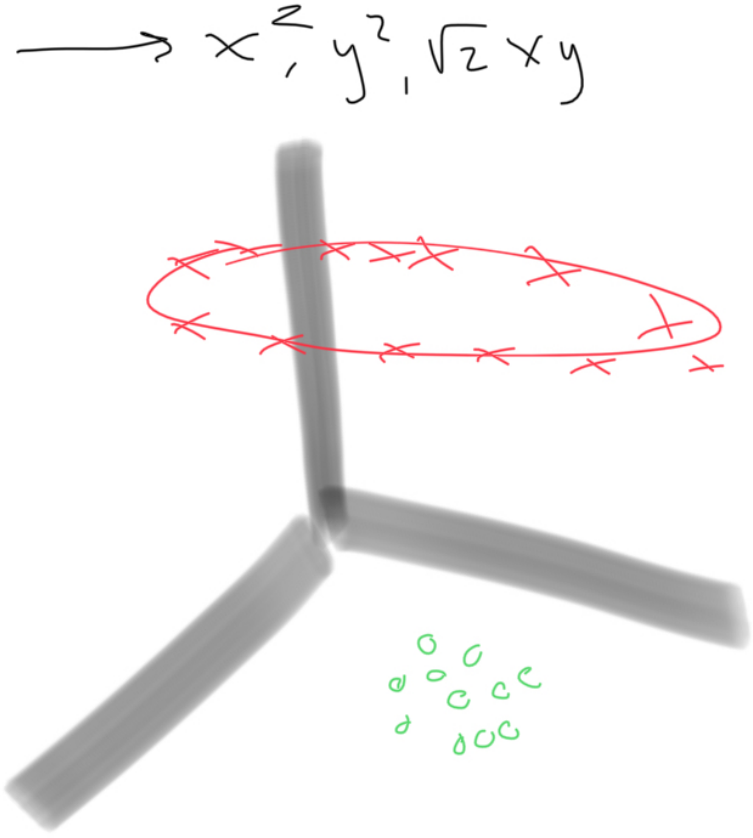

Figure 3.5: Data set with basis functions applied, now linearly separable.

we can write our generalized linear model as:

$$h_k(\mathbf{x}, \mathbf{w}_k) = \mathbf{w}_k^T \phi(\mathbf{x}) = \mathbf{w}_k^T \boldsymbol{\phi} \tag{3.2}$$

For the sake of simplicity in the rest of this chapter, we will leave out any basis changes in our derivations, but you should recognize that they could be applied to any of our input data to make the problems more tractable.

⋆ For an input matrix $\mathbf{X}$, there is a matrix generalization of our basis transformed inputs: $\boldsymbol{\Phi} = \phi(\mathbf{X})$, where $\boldsymbol{\Phi}$ is known as the *design matrix*.

3.4 Algorithms for Decision Boundaries

Now that we have a high-level understanding of what we're trying to accomplish with discriminant functions, we can consider how to solve for the decision boundaries that will dictate our classification decisions. Similar to linear regression, this means we need to establish an objective function to optimize over a training data set. We begin with an objective function that should be familiar from the last chapter: least squares loss.

3.4.1 Least Squares Loss

To find the set of weights $\mathbf{w}$ that form the optimal decision boundary between target classes, we will start with a technique that we also used for linear regression: minimizing a least squares loss function.

We first need to introduce the idea of *one-hot encoding*, which simply means that the class of a given data point is described by a vector with K options that has a 1 in the position that corresponds to class C_k and 0s everywhere else (note that these classes aren't usually 0-indexed). For example, class C_1 of 4 classes would be represented by the vector:

$$\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{3.3}$$

While class C_2 would be represented by the vector:

$$\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \tag{3.4}$$

and so on. Now that we have the idea of one-hot encoding, we can describe our target classes for each data point in terms of a one-hot encoded vector, which can then be used in our training process for least squares.

Each class C_k gets its own linear function with a different set of weights $\mathbf{w}_k$:

$$h_k(\mathbf{x}, \mathbf{w}_k) = \mathbf{w}_k^T \mathbf{x}$$

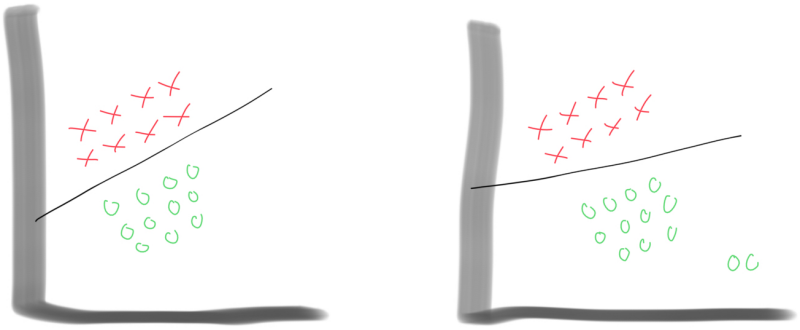

Figure 3.6: Outliers significantly impact our decision boundary.

We can combine the set of weights for each class into a matrix $\mathbf{W}$, which gives us our linear classifier:

$$h(\mathbf{x}, \mathbf{W}) = \mathbf{W}^T\mathbf{x} \tag{3.5}$$

where each row in the weight matrix $\mathbf{W}$ corresponds to the linear function of an individual class. We can use the results derived in the last chapter to find the solution for $\mathbf{W}$ that minimizes the least squares loss function. Assuming a data set of input data points $\mathbf{X}$ and one-hot encoded target vectors $\mathbf{Y}$ (where every row is a single target vector), the optimal solution for $\mathbf{W}$ is given by:

$$\mathbf{W}^* = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{Y}$$

which we can then use in our discriminant function $h(\mathbf{x}, \mathbf{W}^*)$ to make predictions on new data points.

While least squares gives us an analytic solution for our discriminant function, it also has some significant limitations. For one, least squares penalizes data points that are 'too good', meaning they fall too far on the correct side of the decision boundary. Furthermore, it is not robust to outliers, meaning the decision boundary significantly changes with the addition of just a few outlier data points, as seen in Figure 3.6.

We can help remedy the problems with least squares by using alternative methods for solving for our weight parameters.

3.4.2 Fisher's Linear Discriminant

Should we mention this here? Is it necessary?

3.4.3 Perceptron Algorithm

To explain the perceptron algorithm, we must first motivate something called **0/1 loss**, which is simply an alternative loss function to least squares. The idea behind it is very simple: if we misclassify a point, we incur a loss of 1, and if we classify it correctly, we incur no loss.

While this is a very intuitive loss function, it does not have a closed form solution like least squares does, and it is non-convex so it is not easily optimized. However, the 0/1 loss function inspires a different loss function, known as the **perceptron loss function**.

The perceptron loss function is a modification of the 0/1 loss function that both provides more fine-grained information and makes it differentiable (which will be important for our ability to optimize our parameters).

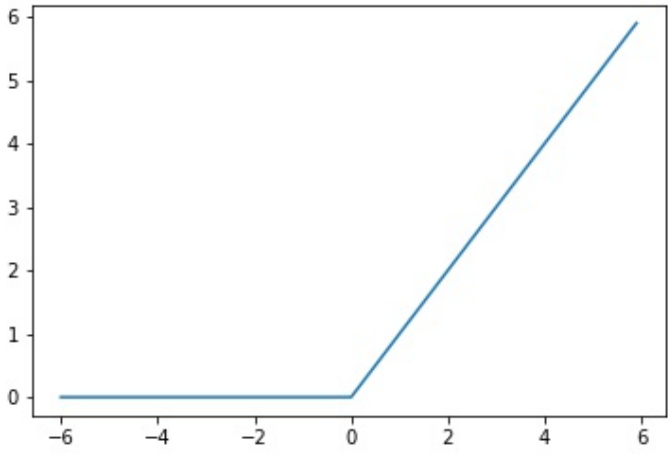

Figure 3.7: Form of the ReLU function.

To understand the perceptron loss function, it's first necessary to introduce the *rectified linear activation unit*, known as ReLU, seen in Figure 3.7.

$$\text{ReLU}(z) = \max\{0, z\} \tag{3.6}$$

We can use the form of this function to our advantage in constructing the perceptron loss by recognizing that we wish to incur error when we're wrong (which corresponds to $z > 0$, the right side of the graph that is continuously increasing), and we wish to incur 0 error if we are correct (which corresponds to the left side of the graph where $z < 0$).

Remember from the previous section on least squares that in the two-class case, we classify a data point $\mathbf{x}^*$ as being from class **1** if $h(\mathbf{x}^*, \mathbf{w}) \geq 0$, and class **-1** otherwise. We can combine this logic with ReLU by recognizing that $-h(\mathbf{x}^*, \mathbf{w})y^* \geq 0$ when there is a classification error, where y^* is the true class of data point $\mathbf{x}^*$. This has exactly the properties we described above: we incur error when we misclassify, and otherwise we do not incur error.

We can then write the entirety of the perceptron loss function:

$$\mathcal{L}(\mathbf{w}) = \sum_{i=1}^{N} \text{ReLU}(-h(\mathbf{x}_i, \mathbf{w})y_i) \tag{3.7}$$

$$= -\sum_{y_i \neq \hat{y}_i}^{N} h(\mathbf{x}_i, \mathbf{w})y_i \tag{3.8}$$

$$= -\sum_{y_i \neq \hat{y}_i}^{N} \mathbf{w}^T \mathbf{x}_i y_i \tag{3.9}$$

where $\hat{y}_i$ is our class prediction and y_i is the true class value. Notice that misclassified examples contribute positive loss, as desired. We can take the gradient of this loss function, which will allow us to optimize it using stochastic gradient descent. The gradient of the loss with respect to our

parameters $\mathbf{w}$ is as follows:

$$\frac{\partial \mathcal{L}(\mathbf{w})}{\partial \mathbf{w}} = -\sum_{y_i \neq \hat{y}_i}^{N} \mathbf{x}_i y_i$$

and then our update equation from time t to time $t+1$ for a single misclassified example and with learning rate η is given by:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \frac{\partial \mathcal{L}(\mathbf{w})}{\partial \mathbf{w}} = \mathbf{w}^{(t)} + \eta \mathbf{x}_i y_i$$

There is no need to understand what a learning rate η is at the moment, just know for now that it's needed in order to efficiently optimize our parameters $\mathbf{w}$. (Should I explain SGD here???) To sum up, the benefits of the perceptron loss function are its differentiability (which allows us to optimize our weight parameters), the fact that it doesn't penalize any correctly classified data points (unlike basic linear classification), and that it penalizes more heavily data points that are more poorly misclassified. Furthermore, the perceptron algorithm guarantees that if there is separability between all of our data points and we run the algorithm for long enough, we will find a setting of parameters that perfectly separates our data set. The proof for this is beyond the scope of this textbook.

3.5 Probabilistic Methods

Unsurprisingly, we can also cast the problem of classification into a probabilistic context, which we now turn our attention to. Within this setting, we have a secondary choice to make between two distinct probabilistic approaches: discriminative or generative. We will explore both of these options.

3.5.1 Probabilistic Discriminative Models

Ultimately, our classification task can be summarized as follows: *given a new data point $\boldsymbol{x}^*$, can we accurately predict the target class y^*?*

Given this problem statement, it makes sense that we might try to model the distribution of $y^*|\mathbf{x}^*$. In fact, modeling this conditional distribution directly is what's known as **probabilistic discriminative modeling**.

Definition 3.5.1 (Probabilistic Discriminative Modeling): Probabilistic modeling is a classification technique whereby we choose to directly model the conditional class distribution in order to make classification predictions.

This means that we will start with the functional form of the generalized linear model described by Equation 3.2, convert this to a conditional distribution, and then optimize the parameters of the conditional distribution directly using a maximum likelihood procedure. From here, we will be able to make predictions on new data points $\mathbf{x}^*$. The key feature of this procedure, which is known as *discriminative training*, is that it optimizes the parameters of a conditional distribution directly. We describe a specific, common example of this type of procedure called **logistic regression** in the next section.

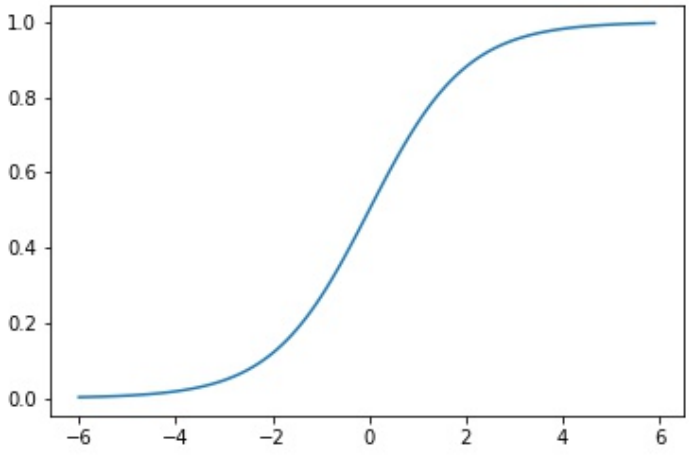

Figure 3.8: Logistic Sigmoid Function.

Logistic Regression

One problem we need to face in our discriminative modeling paradigm is that the results of our generalized linear model are not probabilities; they are simply real numbers. This is why in the previous paragraph we mentioned needing to convert our generalized linear model to a conditional distribution. That step boils down to somehow squashing the outputs of our generalized linear model onto the real numbers between 0 and 1, which will then correspond to probabilities. To do this, we will apply what is known as the **logistic sigmoid function**, $\sigma(\cdot)$.

Definition 3.5.2 (Logistic Sigmoid Function, $\sigma(\cdot)$): The logistic sigmoid function is commonly used to compress the real number line down to values between 0 and 1. It is defined functionally as:

$$\sigma(z) = \frac{1}{1 + \exp(-z)}$$

As you can see in Figure 3.8 where the logistic sigmoid function is graphed, it squashes our output domain between 0 and 1 as desired for a probability.

⋆ There is a more satisfying derivation for our use of the logistic sigmoid function in logistic regression, but understanding its squashing properties as motivation is sufficient for the purposes of this book.

Using the logistic sigmoid function, we now have a means of generating a probability that a new data point $\mathbf{x}^*$ is part of class y^*. Because we are currently operating in the two-class case, which in this context will be denoted C_1 and C_2, we'll write the probability for each of these classes as:

$$p(y^* = C_1|\mathbf{x}^*) = \sigma(\mathbf{w}^T\mathbf{x}^*)$$
$$p(y^* = C_2|\mathbf{x}^*) = 1 - p(y^* = C_1|\mathbf{x}^*)$$

Now that we have such functions, we can apply the maximum likelihood procedure to determine the optimal parameters for our logistic regression model.

For a data set $\{\mathbf{x}_i, y_i\}$ where $i = 1..N$ and $y_i \in \{0, 1\}$, the likelihood for our setting of parameters $\mathbf{w}$ can be written as:

$$p(\{y_i\}|\mathbf{w}) = \prod_{i=1}^{N} \hat{y}_i^{y_i} \{1 - \hat{y}_i\}^{1-y_i} \tag{3.10}$$

where $\hat{y}_i = p(y_i = C_1|\mathbf{x}_i) = \sigma(\mathbf{w}^T\mathbf{x}_i)$.

In general, we would like to maximize this probability to find the optimal setting of our parameters. This is exactly what we intend to do, but with two further simplifications. First, we're going to maximize the probability of the *logarithm* of the likelihood. As a monotonically increasing function, maximizing the logarithm of the likelihood (called the *log likelihood*) will result in the same optimal setting of parameters as if we had just optimized the likelihood directly. Furthermore, using the log likelihood has the nice effect of turning what is currently a product of terms from $1..N$ to a sum of terms from $1..N$, which will make our calculations nicer. Second, we will turn our log likelihood into an *error function* by taking the negative of our log likelihood expression. Now, instead of maximizing the log likelihood, we will be minimizing the error function, which will again find us the same setting of parameters.

⋆ It's worth rereading the above paragraph again to understand the pattern presented there, which we will see several times throughout this book. Instead of maximizing a likelihood function directly, it is often easier to define an error function using the negative log likelihood, which we can then minimize to find the optimal setting of parameters for our model.

After taking the negative logarithm of the likelihood function defined by Equation 3.10, we are left with the following term, known as the *cross-entropy error function*, which we will seek to minimize:

$$\mathrm{E}(\mathbf{w}) = -\ln p(\{y_i\}|\mathbf{w}) = -\sum_{i=1}^{N} \{y_i \ln \hat{y}_i + (1 - y_i) \ln (1 - \hat{y}_i)\} \tag{3.11}$$

where as before $\hat{y}_i = p(y_i = C_1|\mathbf{x}_i) = \sigma(\mathbf{w}^T\mathbf{x}_i)$. Now, to solve for the optimal setting of parameters using a maximum likelihood approach as we've done previously, we start by taking the gradient of the cross-entropy error function with respect to $\mathbf{w}$:

$$\nabla \mathrm{E}(\mathbf{w}) = \sum_{i=1}^{N} (\hat{y}_i - y_i)\mathbf{x}_i \tag{3.12}$$

which we arrive at by recognizing that the derivative of the logistic sigmoid function can be written in terms of itself as:

$$\frac{\partial \sigma(z)}{\partial z} = \sigma(z)(1 - \sigma(z))$$

(I can include the derivation if that's helpful, thoughts???) Let's inspect the form of Equation 3.12 for a moment to understand its implications. First, it's a summation over all of our data points, as we would expect. Then, for each data point, we are taking the difference between our predicted value $\hat{y}_i$ and the actual value y_i, and multiplying that difference by the input vector $\mathbf{x}_i$.

While a closed form solution does not present itself here as it did in the case of linear regression due to the nonlinearity of the logistic sigmoid function, we can still optimize the parameters $\mathbf{w}$ of our model using an iterative procedure that updates our parameters by moving toward the minimum of the error function defined in Equation 3.11.

Multi-Class Logistic Regression and Softmax

As we saw when working with discriminant functions, we also need to account for multi-class problems, which are practically speaking more common than the simple two-class scenario.

In the logistic regression setting (which is a form of *discriminative modeling*, not to be confused with *discriminant functions*), we are now working with probabilities, which is why we introduced the 'probability squashing' sigmoidal function $\sigma(\cdot)$. Note that the sigmoidal function is also sometimes known as the sigmoidal activation function.

Similarly, in the multi-class logistic regression setting, we would like to also have a probability squashing function that generalizes beyond two classes. This generalization of the sigmoidal function is known as **softmax**.

Definition 3.5.3 (Softmax): Softmax is the multi-class generalization of the sigmoidal activation function. It accepts a vector of activations (inputs) and returns a vector of probabilities corresponding to those activations. It is defined as follows:

$$\text{softmax}_k(\mathbf{z}) = \frac{\exp(z_k)}{\sum_{i=1}^{K} \exp(z_i)}, \text{ for all } k$$

Multi-class logistic regression uses softmax over a vector of activations to select the most likely target class for a new data point. It does this by applying softmax and then assigning the new data point to the class with the highest probability.

Example 3.1 (Softmax Example): Consider an example that has three classes: C_1, C_2, C_3. Let's say we have an activation vector $\mathbf{z}$ for our new data point $\mathbf{x}$ that we wish to classify, given by:

$$\mathbf{z} = \mathbf{W}^T\mathbf{x} = \begin{bmatrix} 4 \\ 1 \\ 7 \end{bmatrix}$$

where

$$\mathbf{z}_j = \mathbf{w}_j^T\mathbf{x}$$

Then, using our definition of softmax, we have:

$$\text{softmax}(\mathbf{z}) = \begin{bmatrix} 0.047 \\ 0.002 \\ 0.950 \end{bmatrix}$$

And therefore, we would assign our new data point $\mathbf{x}$ to class C_3, which has the largest activation.

As in the two-class logistic regression case, we now need to solve for the parameters $\mathbf{W}$ of our model, also written as $\{\mathbf{w}_j\}$. Assume we have an observed data set $\{\mathbf{x}_i, \mathbf{y}_i\}$ for $i = 1..N$ where $\mathbf{y}_i$ are one-hot encoded target vectors. We begin this process by writing the likelihood for our data, which is only slightly modified here to account for multiple classes:

$$p(\{\mathbf{y}_i\}|\mathbf{W}) = \prod_{i=1}^{N}\prod_{j=1}^{K} p(\mathbf{y}_i = C_k|\mathbf{x}_i)^{y_{ij}} = \prod_{i=1}^{N}\prod_{j=1}^{K} \hat{y}_{ij}^{y_{ij}} \tag{3.13}$$

where $\hat{y}_{ij} = \text{softmax}_j(\mathbf{W}\mathbf{x}_i)$

We can now take the negative logarithm to get the cross-entropy error function for the multi-class classification problem:

$$\text{E}(\mathbf{W}) = -\ln p(\{\mathbf{y}_i\}|\mathbf{W}) = -\sum_{i=1}^{N}\sum_{j=1}^{K} y_{ij} \ln \hat{y}_{ij} \tag{3.14}$$

As in the two-class case, we now take the gradient with respect to one of our weight parameter vectors $\mathbf{w}_j$:

$$\nabla_{\mathbf{w}_j}\text{E}(\mathbf{W}) = \sum_{i=1}^{N}(\hat{y}_{ij} - y_{ij})\mathbf{x}_i \tag{3.15}$$

which we arrived at by recognizing that the derivative of the softmax function with respect to the input activations z_j can be written in terms of itself:

$$\frac{\partial \text{softmax}_k(z)}{\partial z_j} = \text{softmax}_k(z)(\text{I}_{kj} - \text{softmax}_j(z))$$

where I is the identity matrix.

As in the two-class case, now that we have this gradient expression, we can use an iterative procedure to update our parameters $\mathbf{W}$ toward their optimal values by minimizing the error function.

3.5.2 Probabilistic Generative Models

With the probabilistic discriminative modeling approach, we elected to directly model the class-conditional probability $y^*|\mathbf{x}^*$. However, there was an alternative option: we could have instead modeled the joint distribution of the class y^* and the input data point $\mathbf{x}^*$ together as $p(y^*, \mathbf{x}^*)$. This approach is what's known as **probabilistic generative modeling** because we actually model the process by which the data was generated.

To model the data generating process in classification tasks generally acknowledges that a data point is produced by first selecting a class y^* from a categorical class prior $p(y^*)$ and then generating the data point $\mathbf{x}^*$ itself from the class-conditional distribution $p(y^*|\mathbf{x}^*)$, the form of which is problem specific.

⋆ Notice that with probabilistic generative modeling, we choose a specific distribution for our class-conditional densities instead of simply using a generalized linear model combined with a sigmoid/softmax function as we did in the logistic regression setting. This highlights the difference between discriminative and generative modeling: in the generative setting, we are modeling the production of the data itself instead of simply optimizing the parameters of a more general model that predicts class membership directly.

Classification in the Generative Setting

Now that we're situated in the generative setting, we turn our attention to the actual problem of using our model to predict class membership of new data points $\mathbf{x}^*$.

To perform classification, we will pick the class C_k that maximizes the probability of $\mathbf{x}^*$ being from that class as defined by $p(y^* = C_k|\mathbf{x}^*)$. We can relate this conditional density to the joint

density $p(y^*, \mathbf{x}^*)$ through Bayes' Rule:

$$p(y^* = C_k|\mathbf{x}^*) = \frac{p(y^*, \mathbf{x}^*)}{p(\mathbf{x}^*)} = \frac{p(\mathbf{x}^*|y^* = C_k)p(y^* = C_k)}{p(\mathbf{x}^*)} \propto p(\mathbf{x}^*|y^* = C_k)p(y^* = C_k)$$

where $p(\mathbf{x}^*)$ is a constant that can be ignored as it will be the same for every conditional probability $p(y^* = C_k|\mathbf{x}^*)$.

Recall that the class prior $p(y)$ will always be a categorical distribution (the multi-class generalization of the Bernoulli distribution), while the class-conditional distribution can be specified using prior knowledge of the problem domain. Once we have specified this class conditional distribution, we can solve for the parameters of both that model and the categorical distribution using a maximum likelihood procedure.

Maximum Likelihood Solution

We're going to derive the maximum likelihood solution for the parameters of our probabilistic generative model in the two-class setting, allowing that the multi-class generalization will be a straightforward exercise.

Let's start by assuming a Gaussian conditional distribution for our data $p(\mathbf{x}|y = C_k)$. Given a data set $\{\mathbf{x}_i, y_i\}$ for $i = 1..N$, where $y_i = 1$ corresponds to class C_1 and $y_i = 0$ corresponds to class C_2, we can construct our maximum likelihood solution. Let's first specify our class priors:

$$p(C_1) = \pi$$
$$p(C_2) = 1 - \pi$$

For simplicity, we'll assume a shared covariance matrix $\mathbf{\Sigma}$ between our two classes. Then, for data points $\mathbf{x}_i$ from class C_1, we have:

$$p(\mathbf{x}_i, C_1) = p(C_1)p(\mathbf{x}_i|C_1) = \pi\mathcal{N}(\mathbf{x}_i|\boldsymbol{\mu}_1, \mathbf{\Sigma})$$

And for data points $\mathbf{x}_i$ from class C_2, we have:

$$p(\mathbf{x}_i, C_2) = p(C_2)p(\mathbf{x}_i|C_2) = (1 - \pi)\mathcal{N}(\mathbf{x}_i|\boldsymbol{\mu}_2, \mathbf{\Sigma})$$

Using these two densities, we can construct our likelihood function:

$$\mathcal{L}(\pi, \boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \mathbf{\Sigma}) = \prod_{i=1}^{N} \Big(\pi\mathcal{N}(\mathbf{x}_i|\boldsymbol{\mu}_1, \mathbf{\Sigma})\Big)^{y_i} \Big((1 - \pi)\mathcal{N}(\mathbf{x}_i|\boldsymbol{\mu}_2, \mathbf{\Sigma})\Big)^{1-y_i}$$

As usual, we will take the logarithm which is easier to work with:

$$\ln \mathcal{L}(\pi, \boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \mathbf{\Sigma}) = \sum_{i=1}^{N} y_i \ln \Big(\pi\mathcal{N}(\mathbf{x}_i|\boldsymbol{\mu}_1, \mathbf{\Sigma})\Big) + (1 - y_i) \ln \Big((1 - \pi)\mathcal{N}(\mathbf{x}_i|\boldsymbol{\mu}_2, \mathbf{\Sigma})\Big)$$

We can now optimize for our parameters $\pi, \boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \text{and}, \mathbf{\Sigma}$ separately, using the usual procedure of taking the derivative, setting equal to 0, and then solving for the parameter of interest.

Derivation 3.5.1 (MLE Solution): Beginning with π, we'll concern ourselves only with the

terms that depend on π which are:

$$\sum_{i=1}^{N} y_i \ln \pi + (1 - y_i) \ln (1 - \pi)$$

Taking the derivative with respect to π, setting equal to 0, rearranging, we get:

$$\pi = \frac{1}{N} \sum_{i=1}^{N} y_i = \frac{N_1}{N} = \frac{N_1}{N_1 + N_2}$$

where N_1 is the number of data points in our data set from class C_1, N_2 is the number of data points from class C_2, and N is just the total number of data points. This means that the maximum likelihood solution for π is the fraction of points that are assigned to class C_1, a fairly intuitive solution and one that will be commonly seen when working with maximum likelihood calculations. Let's now perform the maximization for $\boldsymbol{\mu}_1$. Start by considering the terms from our log likelihood involving $\boldsymbol{\mu}_1$:

$$\sum_{i=1}^{N} y_i \ln \mathcal{N}(\mathbf{x}_i | \boldsymbol{\mu}_1, \boldsymbol{\Sigma}) = -\frac{1}{2} \sum_{i=1}^{N} y_i (\mathbf{x}_i - \boldsymbol{\mu}_1)^T \boldsymbol{\Sigma}^{-1} (\mathbf{x}_i - \boldsymbol{\mu}_1) + c$$

where c are constants not involving the $\boldsymbol{\mu}_1$ term. Taking the derivative with respect to $\boldsymbol{\mu}_1$, setting equal to 0, and rearranging:

$$\boldsymbol{\mu}_1 = \frac{1}{N_1} \sum_{i=1}^{N} y_i \mathbf{x}_i$$

which is simply the average of all the data points $\mathbf{x}_i$ assigned to class C_1, a very intuitive result. By the same derivation, the maximum likelihood solution for $\boldsymbol{\mu}_2$ is:

$$\boldsymbol{\mu}_2 = \frac{1}{N_2} \sum_{i=1}^{N} (1 - y_i) \mathbf{x}_i$$

And finally, we can derive the maximum likelihood solution for the shared covariance matrix $\boldsymbol{\Sigma}$. Start by considering the terms in our log likelihood expression involving $\boldsymbol{\Sigma}$:

$$-\frac{1}{2} \sum_{i=1}^{N} y_i \ln |\boldsymbol{\Sigma}| - \frac{1}{2} \sum_{i=1}^{N} y_i (\mathbf{x}_i - \boldsymbol{\mu}_1)^T \boldsymbol{\Sigma}^{-1} (\mathbf{x}_i - \boldsymbol{\mu}_1) - \frac{1}{2} \sum_{i=1}^{N} (1 - y_i) \ln |\boldsymbol{\Sigma}| - \frac{1}{2} \sum_{i=1}^{N} (1 - y_i) (\mathbf{x}_i - \boldsymbol{\mu}_2)^T \boldsymbol{\Sigma}^{-1} (\mathbf{x}_i - \boldsymbol{\mu}_2)$$

Taking the derivative with respect to $\boldsymbol{\Sigma}$:

$$N \boldsymbol{\Sigma}^{-T} - \frac{1}{2} \sum_{i=1}^{N} y_i \boldsymbol{\Sigma}^{-T} (\mathbf{x}_i - \boldsymbol{\mu}_1)(\mathbf{x}_i - \boldsymbol{\mu}_1)^T \boldsymbol{\Sigma}^{-T} - \frac{1}{2} \sum_{i=1}^{N} (1 - y_i) \boldsymbol{\Sigma}^{-T} (\mathbf{x}_i - \boldsymbol{\mu}_2)(\mathbf{x}_i - \boldsymbol{\mu}_2)^T \boldsymbol{\Sigma}^{-T}$$

Setting equal to 0 and rearranging to solve for $\boldsymbol{\Sigma}$:

$$\boldsymbol{\Sigma} = \frac{1}{N} \sum_{i=1}^{N} \left(y_i (\mathbf{x}_i - \boldsymbol{\mu}_1)(\mathbf{x}_i - \boldsymbol{\mu}_1)^T + (1 - y_i)(\mathbf{x}_i - \boldsymbol{\mu}_2)(\mathbf{x}_i - \boldsymbol{\mu}_2)^T \right)$$

which has the intuitive interpretation that the maximum likelihood solution for the shared covariance matrix is the weighted average of the two individual covariance matrices.

It is relatively straightforward to extend these maximum likelihood derivations from their two-class form to their more general, multi-class form.

Naive Bayes

There exists a further simplification to probabilistic generative modeling known as **Naive Bayes**.

Definition 3.5.4 (Naive Bayes): Naive Bayes is a type of generative model for classification tasks. It imposes the simplifying rule that for a given class C_k, we assume that each feature of the data points $\mathbf{x}$ generated within that class are independent (hence the descriptor 'naive'). This means that the conditional distribution $p(\mathbf{x}|y = C_k)$ can be written as:

$$p(\mathbf{x}|y = C_k) = \prod_{i=1}^{D} p(x_j|y = C_k)$$

where D is the number of features in our data point $\mathbf{x}$ and C_k is the class. Note that Naive Bayes does not specify the form of the model $p(x_j|y = C_k)$, this decision is left up to us.

This is obviously not a realistic simplification for all scenarios, but it can make our calculations easier and may actually hold true in certain cases. We can build more intuition for how Naive Bayes works through an example.

Example 3.2 (Naive Bayes Example): Suppose you are given a biased two-sided coin and two biased dice. The coin has probabilities as follows:

Heads : 30%
Tails : 70%

The dice have the numbers 1 through 6 on them, but they are biased differently. Die 1 has probabilities as follows:

1 : 40%
2 : 20%
3 : 10%
4 : 10%
5 : 10%
6 : 10%

Die 2 has probabilities as follows:

1 : 20%
2 : 20%
3 : 10%
4 : 30%
5 : 10%
6 : 10%

Your friend is tasked with doing the following. First, they flip the coin. If it lands Heads, they select Die 1, otherwise they select Die 2. Then, they roll that die 10 times in a row, recording the results of the die rolls. After they have completed this, you get to observe the aggregated results from the die rolls. Using this information (and assuming you know the biases associated with the coin and dice), you must then classify which die the rolls came from. Assume your friend went through this procedure and produced the following counts:

$$\begin{aligned}
&\mathbf{1}: 3\\
&\mathbf{2}: 1\\
&\mathbf{3}: 2\\
&\mathbf{4}: 2\\
&\mathbf{5}: 1\\
&\mathbf{6}: 1
\end{aligned}$$

Determine which die this roll count most likely came from.

Solution:
This problem is situated in the Naive Bayes framework: for a given class (dictated by the coin flip), the outcomes within that class (die rolls) are independent. Making a classification in this situation is as simple as computing the probability that the selected die produced the given roll counts. Let's start by computing the probability for Die 1:

$$\begin{aligned}
p(\text{Die 1}) &= p(\text{Coin Flip} = \text{Heads}) * p(\text{Roll Count} = [3,1,2,2,1,1])\\
&\propto 0.3 * (0.4)^3 * (0.2)^1 * (0.1)^2 * (0.1)^2 * (0.1)^1 * (0.1)^1\\
&\propto 3.84 * 10^{-9}
\end{aligned}$$

Notice that we don't concern ourselves with the normalization constant for the probability of the roll count - this will not differ between the choice of dice and we can thus ignore it for simplicity. Now the probability for Die 2:

$$\begin{aligned}
p(\text{Die 2}) &= p(\text{Coin Flip} = \text{Tails}) * p(\text{Roll Count} = [3,1,2,2,1,1])\\
&\propto 0.7 * (0.2)^3 * (0.2)^1 * (0.1)^2 * (0.3)^2 * (0.1)^1 * (0.1)^1\\
&\propto 1.008 * 10^{-8}
\end{aligned}$$

Therefore, we would classify this roll count as having come from Die 2.

Note that this problem asked us only to make a classification prediction after we already knew the parameters governing the coin flip and dice rolls. However, given a data set, we could have also used a maximum likelihood procedure under the Naive Bayes assumption to estimate the values of the parameters governing the probability of the coin flip and die rolls.

3.6 Conclusion

In this chapter, we looked at different techniques for solving classification problems, including discriminant functions, probabilistic discriminative models, and probabilistic generative models. In particular, we emphasized the distinction between two-class and multi-class problems as well as the philosophical differences between generative and discriminative modeling.

We also covered several topics that we will make use of in subsequent chapters, including sigmoid functions and softmax, maximum likelihood solutions, and further use of basis changes.

By now, you have a sound understanding of generative modeling and how it can be applied to classification tasks. In the next chapter, we will explore how generative modeling is applied to a still broader class of problems.

3.7 Practice Problems

1. Small toy example of solving for optimal weights for discriminant functions.
2. Demonstrate that least squares penalizes points that are 'too good'.
3. One update step of the perceptron algorithm.
4. Using softmax.
5. Naive Bayes problem - estimating the parameters given a data set derivation.
6. Maximum likelihood estimation for probabilistic generative models in the multi-class setting.
7. Using probabilistic discriminative modeling and probabilistic generative modeling to make predictions on a small data set.