Lectures on Machine Learning

Jeremy Teitelbaum

12/1/22

Table of contents

Preface

This is a Quarto book.

To learn more about Quarto books visit https://quarto.org/docs/books.

1 Linear Regression

1.1 Introduction

Suppose that we are trying to study two quantities x and y that we suspect are related – at least approximately – by a linear equation $y = ax + b$. Sometimes this linear relationship is predicted by theoretical considerations, and sometimes it is just an empirical hypothesis.

For example, if we are trying to determine the velocity of an object travelling towards us at constant speed, and we measure measure the distances $d_1, d_2, \ldots, d_n$ between us and the object at a series of times $t_1, t_2, \ldots, t_n$, then since "distance equals rate times time" we have a theoretical foundation for the assumption that $d = rt + b$ for some constants r and b. On the other hand, because of unavoidable experimental errors, we can't expect that this relationship will hold exactly for the observed data; instead, we likely get a graph like that shown in Figure 1.1. We've drawn a line on the plot that seems to capture the true slope (and hence velocity) of the object.

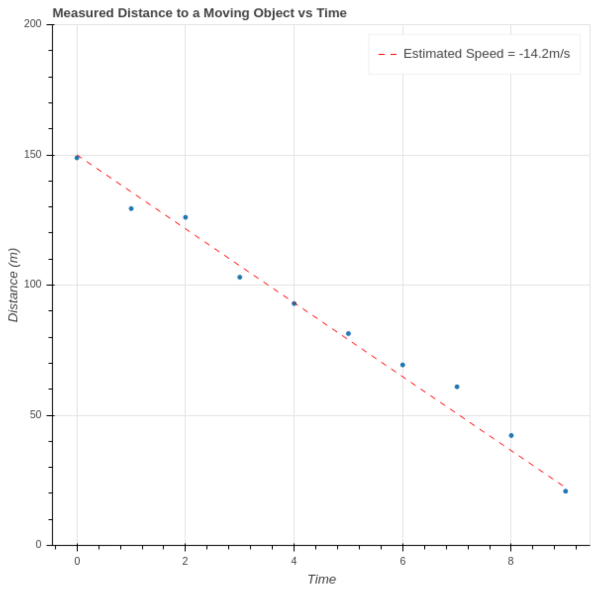


Figure 1.1: Physics Experiment

On the other hand, we might look at a graph such as Figure 1.2, which plots the gas mileage of various car models against their engine size (displacement), and observe a general trend in which bigger engines get lower mileage. In this situation we could ask for the best line of the form $y = mx + b$ that captures this relationship and use that to make general conclusions without necessarily having an underlying theory.

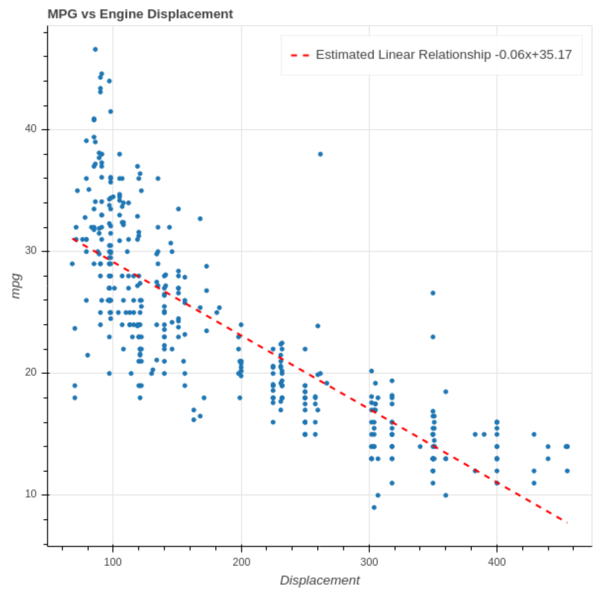


Figure 1.2: MPG vs Displacement ([1])

1.2 Least Squares (via Calculus)

In either of the two cases above, the question we face is to determine the line $y = mx + b$ that "best fits" the data $\{(x_i, y_i)_{i=1}^N\}$. The classic approach is to determine the equation of a line $y = mx + b$ that minimizes the "mean squared error":

$$MSE(m, b) = \frac{1}{N} \sum_{i=1}^{n} (y_i - mx_i - b)^2$$

It's worth emphasizing that the MSE is a function of two variables – the slope m and the intercept b – and that the data points $\{(x_i, y_i)\}$ are constants for these purposes. Furthermore, it's a quadratic function in those two variables. Since our goal is to find m and b that minimize the MSE, we have a Calculus problem that we can solve by taking partial derivatives and setting them to zero.

To simplify the notation, let's abbreviate MSE by E.

$$\frac{\partial E}{\partial m} = \frac{1}{N}\sum_{1}^{N} -2x_i(y_i - mx_i - b)$$
$$\frac{\partial E}{\partial b} = \frac{1}{N}\sum_{1}^{N} -2(y_i - mx_i - b)$$

We set these two partial derivatives to zero, so we can drop the -2 and regroup the sums to obtain two equations in two unknowns (we keep the $\frac{1}{N}$ because it is illuminating in the final result):

$$\begin{aligned} \frac{1}{N}(\sum_{i=1}^{N} x_i^2)m + \quad \frac{1}{N}(\sum_{i=1}^{N} x_i)b &= \quad \frac{1}{N}\sum_{i=1}^{N} x_i y_i \\ \frac{1}{N}(\sum_{i=1}^{N} x_i)m + \qquad\qquad b &= \quad \frac{1}{N}\sum_{i=1}^{N} y_i \end{aligned} \tag{1.1}$$

In these equations, notice that $\frac{1}{N}\sum_{i=1}^{N} x_i$ is the average (or mean) value of the x_i. Let's call this $\overline{x}$. Similarly, $\frac{1}{N}\sum_{i=1}^{N} y_i$ is the mean of the y_i, and we'll call it $\overline{y}$. If we further simplify the notation and write S_{xx} for $\frac{1}{N}\sum_{i=1}^{N} x_i^2$ and S_{xy} for $\frac{1}{N}\sum_{i=1}^{N} x_i y_i$ then we can write down a solution to this system using Cramer's rule:

$$\begin{aligned} m &= \frac{S_{xy} - \overline{xy}}{S_{xx} - \overline{x}^2} \\ b &= \frac{S_{xx}\overline{y} - S_{xy}\overline{x}}{S_{xx} - \overline{x}^2} \end{aligned} \tag{1.2}$$

where we must have $S_{xx} - \overline{x}^2 \neq 0$.

1.2.1 Exercises

1. Verify that Equation 1.2 is in fact the solution to the system in Equation 1.1 .
2. Suppose that $S_{xx} - \overline{x}^2 = 0$. What does that mean about the x_i? Does it make sense that the problem of finding the "line of best fit" fails in this case?

1.3 Least Squares (via Geometry)

In our discussion above, we thought about our data as consisting of N pairs (x_i, y_i) corresponding to n points in the xy-plane $\mathbf{R}^2$. Now let's turn that picture "on its side", and instead think of our data as consisting of *two* points in $\mathbf{R}^n$:

$$X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \text{ and } Y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

Let's also introduce one other vector

$$E = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}.$$

First, let's assume that E and X are linearly independent. If not, then X is a constant vector (why?) which we already know is a problem from Section 1.2, Exercise 2. Therefore E and X span a plane in $\mathbf{R}^n$.

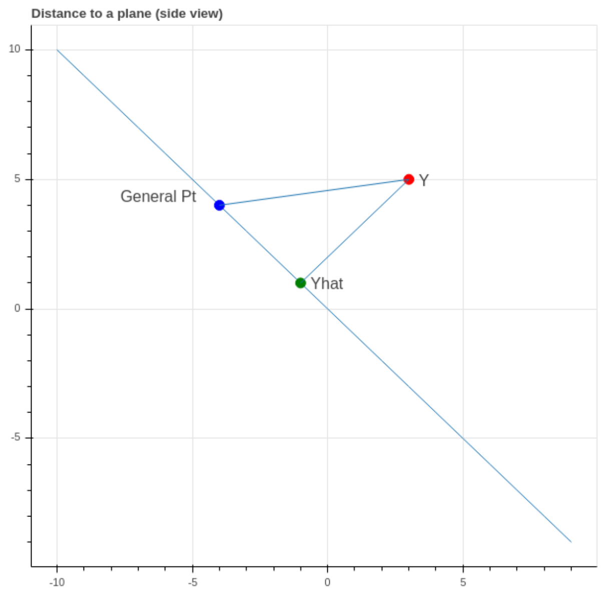


Figure 1.3: Distance to A Plane

Now if our data points (x_i, y_i) all *did* lie on a line $y = mx + b$, then the three vectors X, Y, and E would be linearly dependent:

$$Y = mX + bE.$$

Since our data is only approximately linear, that's not the case. So instead we look for an approximate solution. One way to phrase that is to ask:

What is the point $\hat{Y}$ in the plane H spanned by X and E in $\mathbf{R}^n$ which is closest to Y?

If we knew this point $\hat{Y}$, then since it lies in H we would have $\hat{Y} = mX+bE$ and the coefficients m and b would be a candidate for defining a line of best fit $y = mx + b$. Finding the point in a plane closest to another point in $\mathbf{R}^n$ is a geometry problem that we can solve.

Proposition: The point $\hat{Y}$ in the plane spanned by X and E is the point such that the vector $Y - \hat{Y}$ is perpendicular to H.

Proof: See Figure 1.3 for an illustration – perhaps you are already convinced by this, but let's be careful. $\hat{Y} = mX + bE$ such that

$$D = \|Y - \hat{Y}\|^2 = \|Y - mX - bE\|^2$$

is minimal. Using some vector calculus, we have

$$\frac{\partial D}{\partial m} = \frac{\partial}{\partial m}(Y - mX - bE) \cdot (Y - mX - bE) = -2(Y - mX - bE) \cdot X$$

and

$$\frac{\partial D}{\partial b} = \frac{\partial}{\partial b}(Y - mX - bE) \cdot (Y - mX - bE) = -2(Y - mX - bE) \cdot E.$$

So both derivatives are zero exactly when $\hat{Y} = (Y - mX - bE)$ is orthogonal to both X and E, and therefore every vector in H.

We also obtain equations for m and b just as in our first look at this problem.

$$\begin{aligned} m(X \cdot E) + b(E \cdot E) &= (Y \cdot E) \\ m(X \cdot X) + b(E \cdot X) &= (Y \cdot X) \end{aligned} \tag{1.3}$$

We leave it is an exercise below to check that these are the same equations that we obtained in Equation 1.2.

1.3.1 Exercises

1. Verify that Equation 1.2 and Equation 1.3 are equivalent.

1.4 The Multivariate Case (Calculus)

Having worked through the problem of finding a "line of best fit" from two points of view, let's look at a more general problem. We looked above at a scatterplot showing the relationship between gas mileage and engine size (displacement). There are other factors that might contribute to gas mileage that we want to consider as well – for example:

- a car that is heavy compared to its engine size may get worse mileage
- a sports car with a drive train that gives fast acceleration as compared to a car with a transmission designed for long trips may have different mileage for the same engine size.

Suppose we wish to use engine displacement, vehicle weight, and acceleration all together to predict mileage. Instead of looking points (x_i, y_i) where x_i is the displacement of the i^{th} car model and we try to predict a value y from a corresponding x as $y = mx + b$ – let's look at a situation in which our measured value y depends on multiple variables – say displacement d, weight w, and acceleration a with $k = 3$ – and we are trying to find the best linear equation

$$y = m_1 d + m_2 w + m_3 a + b \tag{1.4}$$

But to handle this situation more generally we need to adopt a convention that will allow us to use indexed variables instead of d, w, and a. We will use the *tidy* data convention.

Tidy Data: A dataset is tidy if it consists of values x_{ij} for $i = 1, \ldots, N$ and $j = 1, \ldots, k$ so that:

- the row index corresponds to a *sample* – a set of measurements from a single event or item;
- the column index corresponds to a *feature* – a particular property measured for all of the events or items.

In our case,

- the *samples* are the different types of car models,
- the *features* are the properties of those car models.

For us, N is the number of different types of cars, and k is the number of properties we are considering. Since we are looking at displacement, weight, and acceleration, we have $k = 3$.

So the "independent variables" for a set of data that consists of N samples, and k measurements for each sample, can be represented by a $N \times k$ matrix

$$X = \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1k} \\ x_{21} & x_{22} & \cdots & x_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N1} & x_{k2} & \cdots & x_{Nk} \end{pmatrix}$$

and the measured dependent variables Y are a column vector

$$Y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix}.$$

If $m_1, \ldots, m_k$ are "slopes" associated with these properties in Equation 1.4, and b is the "intercept", then the predicted value $\hat{Y}$ is given by a matrix equation

$$\hat{Y} = X \begin{bmatrix} m_1 \\ m_2 \\ \cdots \\ m_k \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ \cdots \\ 1 \end{bmatrix} b$$

and our goal is to choose these parameters m_i and b to make the mean squared error:

$$MSE(m_1, \ldots, m_k, b) = \|Y - \hat{Y}\|^2 = \sum_{i=1}^{N} (y_i - \sum_{j=1}^{k} x_{ij} m_j - b)^2.$$

Here we are summing over the N different car models, and for each model taking the squared difference between the true mileage y_i and the "predicted" mileage $\sum_{j=1}^{k} x_{ij} m_j + b$. We wish to minimize this MSE.

Let's make one more simplification. The intercept variable b is annoying because it requires separate treatment from the m_i. But we can use a trick to eliminate the need for special treatment. Let's add a new feature to our data matrix (a new column) that has the constant value 1.

$$X = \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1k} & 1 \\ x_{21} & x_{22} & \cdots & x_{2k} & 1 \\ \vdots & \vdots & \ddots & \vdots & 1 \\ x_{N1} & x_{k2} & \cdots & x_{Nk} & 1 \end{pmatrix}$$

Now our data matrix X is $N \times (k+1)$ and we can put our "intercept" $b = m_{k+1}$ into our vector of "slopes" $m_1, \ldots, m_k, m_{k+1}$:

$$\hat{Y} = X \begin{bmatrix} m_1 \\ m_2 \\ \cdots \\ m_k \\ m_{k+1} \end{bmatrix}$$

and our MSE becomes

$$MSE(M) = \|Y - XM\|^2$$

where

$$M = \begin{bmatrix} m_1 \\ m_2 \\ \cdots \\ m_k \\ m_{k+1} \end{bmatrix}.$$

Remark: Later on (see {Section 1.6}) we will see that if we "center" our features about their mean, by subtracting the average value of each column of X from that column; and we also subtract the average value of Y from the entries of Y, then the b that emerges from the least squares fit is zero. As a result, instead of adding a column of 1's, you can change coordinates to center each feature about its mean, and keep your X matrix $N \times k$.

The Calculus approach to minimizing the MSE is to take its partial derivatives with respect to the m_i and set them to zero. Let's first work out the derivatives in a nice form for later.

Proposition: The gradient of $MSE(M) = E$ is given by

$$nablaE = \begin{bmatrix} \frac{\partial}{\partial M_1}E \\ \frac{\partial}{\partial M_2}E \\ \vdots \\ \frac{\partial}{\partial m_{M+1}}E \end{bmatrix} = -2X^\top Y + 2X^\top XM \tag{1.5}$$

where $X^\top$ is the transpose of X.

Proof: First, remember that the ij entry of $X^\top$ is the ji entry of X. Also, we will use the notation $X[j,:]$ to mean the j^{th} row of X and $X[:,i]$ to mean the i^{th} column of X. (This is copied from the Python programming language; the ':' means that index runs over all possibilities).

Since

$$E = \sum_{j=1}^{N}(Y_j - \sum_{s=1}^{k+1} X_{js}M_s)^2$$

we compute:

$$\begin{aligned}\frac{\partial}{\partial M_t}E &= -2\sum_{j=1}^{N} X_{jt}(Y_j - \sum_{s=1}^{k+1} X_{js}M_s) \\ &= -2(\sum_{j=1}^{N} Y_j X_{jt} - \sum_{j=1}^{N}\sum_{s=1}^{k+1} X_{jt}X_{js}M_s) \\ &= -2(\sum_{j=1}^{N} X_{tj}^{\top} Y_j - \sum_{j=1}^{N}\sum_{s=1}^{k+1} X_{tj}^{\top}X_{js}M_s) \\ &= -2(X^{\top}[t,:]Y - \sum_{s=1}^{k+1}\sum_{j=1}^{N} X_{tj}^{\top}X_{js}M_s) \\ &= -2(X^{\top}[t,:]Y - \sum_{s=1}^{k+1}(X^{\top}X)_{ts}M_s) \\ &= -2[X^{\top}[t,:]Y - (X^{\top}X](t,:)M)\end{aligned} \tag{1.6}$$

Stacking up the different rows to make E yields the desired formula.

Proposition: Assume that $D = X^{\top}X$ is invertible (notice that it is a $(k+1)\times(k+1)$ square matrix so this makes sense). The solution M to the multivariate least squares problem is

$$M = D^{-1}X^{\top}Y \tag{1.7}$$

and the “predicted value” $\hat{Y}$ for Y is

$$\hat{Y} = XD^{-1}X^{\top}Y. \tag{1.8}$$

1.5 The Multivariate Case (Geometry)

Let’s look more closely at the equation obtained by setting the gradient of the error, Equation 1.5, to zero. Remember that M is the unknown vector in this equation, everything else is known:

$$X^{\top}Y = X^{\top}XM$$

Here is how to think about this:

1. As M varies, the $N\times 1$ matrix XM varies over the space spanned by the columns of the matrix X. So as M varies XM is a general element of the subspace H of R^N spanned by the $k+1$ columns of X.

2. The product $X^\top XM$ is a $(k+1)\times 1$ matrix. Each entry is the dot product of the general element of H with one of the $k+1$ basis vectors of H.
3. The product $X^\top Y$ is a $(k+1)\times 1$ matrix whose entries are the dot product of the basis vectors of H with Y.

Therefore, this equation asks for us to find M so that the vector XM in H has the same dot products with the basis vectors of H as Y does. The condition

$$X^\top \cdot (Y - XM) = 0$$

says that $Y - XM$ is orthogonal to H. This argument establishes the following proposition.

Proposition: Just as in the simple one-dimensional case, the predicted value $\hat{Y}$ of the least squares problem is the point in H closest to Y – or in other words the point $\hat{Y}$ in H such that $Y - \hat{Y}$ is perpendicular to H.

1.5.1 Orthogonal Projection

Recall that we introduced the notation $D = X^\top X$, and let's assume, for now, that D is an invertible matrix. We have the formula (see Equation 1.8):

$$\hat{Y} = XD^{-1}X^\top Y.$$

Proposition: The matrix $P = XD^{-1}X^\top$ is an $N \times N$ matrix called the orthogonal projection operator onto the subspace H spanned by the columns of X. It has the following properties:

- PY belongs to the subspace H for any $Y \in \mathbf{R}^N$.
- $(Y - PY)$ is orthogonal to H.
- $P * P = P$.

Proof: First of all, $PY = XD^{-1}X^\top Y$ so PY is a linear combination of the columns of X and is therefore an element of H. Next, we can compute the dot product of PY against a basis of H by computing

$$X^\top PY = X^\top XD^{-1}X^\top Y = X^\top Y$$

since $X^\top X = D$. This equation means that $X^\top(Y - PY) = 0$ which tells us that $Y - PY$ has dot product zero with a basis for H. Finally,

$$PP = XD^{-1}X^\top XD^{-1}X^\top = XD^{-1}X^\top = P.$$

It should be clear from the above discussion that the matrix $D = X^\top X$ plays an important role in the study of this problem. In particular it must be invertible or our analysis above breaks down. In the next section we will look more closely at this matrix and what information it encodes about our data.

1.6 Centered coordinates

Recall from last section that the matrix $D = X^\top X$ is of central importance to the study of the multivariate least squares problem. Let's look at it more closely.

Lemma: The i, j entry of D is the dot product

$$D_{ij} = X[:, i] \cdot X[:, j]$$

of the i^{th} and j^{th} columns of X.

Proof: In the matrix multiplication $X^\top X$, the i^{th} row of $X^\top$ gets "dotted" with the j^{th} column of X to product the i, j entry. But the i^{th} row of $X^\top$ is the i^{th} column of X, as asserted in the statement of the lemma.

A crucial point in our construction above relied on the matrix D being invertible. The following Lemma shows that D fails to be invertible only when the different features (the columns of X) are linearly dependent.

Lemma: D is not invertible if and only if the columns of X are linearly dependent.

Proof: If the columns of X are linearly dependent, then there is a nonzero vector m so that $Xm = 0$. In that case clearly $Dm = X^\top Xm = 0$ so D is not invertible. Suppose D is not invertible. Then there is a nonzero vector m with $Dm = X^\top Xm = 0$. This means that the vector Xm is orthogonal to all of the columns of X. Since Xm belongs to the span H of the columns of X, if it is orthogonal to H it must be zero.

In fact, the matrix D captures some important statistical measures of our data, but to see this clearly we need to make a slight change of basis. First recall that $X[:, k+1]$ is our column of all 1, added to handle the intercept. As a result, the dot product $X[:, i] \cdot X[:, k+1]$ is the sum of the entries in the i^{th} column, and so if we let μ_i denote the average value of the entries in column i, we have

$$\mu_i = \frac{1}{N}(X[:, i] \cdot X[:, k+1])$$

Now change the matrix X by elementary column operations to obtain a new data matrix X_0 by setting

$$X_0[:, i] = X[:, i] - \frac{1}{N}(X[:, i] \cdot X[:, k+1])X[:, k+1] = X[:, i] - \mu_i X[:, k+1]$$

for $i = 1, \ldots, k$.

In terms of the original data, we are changing the measurement scale of the data so that each feature has average value zero, and the subspace H spanned by the columns of X_0 is the same as that spanned by the columns of X. Using X_0 instead of X for our least squares problem, we get

$$\hat{Y} = X_0 D_0^{-1} X_0^\top Y$$

and

$$M_0 = D_0^{-1} X_0^\top Y$$

where $D_0 = X_0^\top X_0$.

Proposition: The matrix D_0 has a block form. Its upper left block is a $k \times k$ symmetric block with entries

$$(D_0)_{ij} = (X[:, i] - \mu_i X[:, k+1]) \cdot (X[:, j] - \mu_j X[:, k+1])$$

Its $(k+1)^{st}$ row and column are all zero, except for the $(k+1), (k+1)$ entry, which is N.

Proof: This follows from the fact that the last row and column entries are (for $i \neq k+1$):

$$(X[:, i] - \mu_i X[:, k+1]) \cdot X[:, k+1] = (X[:, i] \cdot X[:, k+1]) - N\mu_i = 0$$

and for $i = k+1$ we have $X[:, k+1] \cdot X[:, k+1] = N$ since that column is just N 1's.

Proposition: If the x coordinates (the features) are centered so that they have mean zero, then the intercept b is

$$\overline{Y} = \frac{1}{N} \sum y_i.$$

Proof: By centering the coordinates, we replace the matrix X by X_0 and D by D_0. and we are trying to minimize $\|Y - X_0 M_0\|^2$. Use the formula from Equation 1.7 to see that

$$M_0 = D_0^{-1} X_0^\top Y.$$

The b value we are interested in is the last entry m_{k+1} in M_0. From the block form of D_0, we know that D_0^{-1} has bottom row and last column zero except for $1/N$ in position $(k+1) \times (k+1)$. Also $X_0^\top$ has last row consisting entirely of 1. So the bottom entry of $X_0^\top Y$ is $\sum_{i=1}^N y_i$, and the bottom entry b of $D_0^{-1} X_0^\top Y$ is

$$\mu_Y = \frac{1}{N} \sum_{i=1}^{N} y_i.$$

as claimed.

Corollary: If we make a further change of coordinates to define

$$Y_0 = Y - \mu_Y \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}$$

then the associated b is zero. As a result we can forget about the extra column of $1's$ that we added to X to account for it and reduce the dimension of our entire problem by 1.

Just to recap, if we center our data so that $\mu_Y = 0$ and $\mu_i = 0$ for $i = 1, \dots, k$, then the least squares problem reduces to minimizing

$$E(M) = \|Y - XM\|^2$$

where X is the $N \times k$ matrix with j^{th} row $(x_{j1}, x_{j2}, \dots, x_{jk})$ for $j = 1, \dots, N$ and the solutions are as given in Equation 1.7 and Equation 1.8.

1.7 Caveats about Linear Regression

1.7.1 Basic considerations

Reflecting on our long discussion up to this point, we should take note of some of the potential pitfalls that lurk in the use of linear regression.

1. When we apply linear regression, we are explicitly assuming that the variable Y is associated to X via linear equations. This is a big assumption!
2. When we use multilinear regression, we are assuming that changes in the different features have independent effects on the target variable y. In other words, suppose that $y = ax_1 + bx_2$. Then an increase of x_1 by 1 increases y by a, and an increase of x_2 by 1 increases y by b. These effects are independent of one another and combine to yield an increase of $a + b$.
3. We showed in our discussion above that linear regression problem has a solution when the matrix $D = X^\top X$ is invertible, and this happens when the columns of D are linearly independent. When working with real data, which is messy, we could have a situation in which the features we are studying are, in fact, dependent – but because of measurement error, the samples that we collected aren't. In this case, the matrix D will be "close" to being non-invertible, although formally still invertible. In this case, computing D^{-1} leads to numerical instability and the solution we obtain is very unreliable.

1.7.2 Simpson's Effect

Simpson's effect is a famous phenomenon that illustrates that linear regression can be very misleading in some circumstances. It is often a product of "pooling" results from multiple experiments. Suppose, for example, that we are studying the relationship between a certain measure of blood chemistry and an individual's weight gain or less on a particular diet. We do our experiments in three labs, the blue, green, and red labs. Each lab obtains similar results – higher levels of the blood marker correspond to greater weight gain, with a regression line of slope around 1. However, because of differences in the population that each lab is studying, some populations are more susceptible to weight gain and so the red lab sees a mean increase of almost 9 lbs while the blue lab sees a weight gain of only 3 lbs on average.

The three groups of scientists pool their results to get a larger sample size and do a new regression. Surprise! Now the regression line has slope -1.6 and increasing amounts of the marker seem to lead to *less* weight gain!

This is called Simpson's effect, or Simpson's paradox, and it shows that unknown factors (confounding factors) may cause linear regression to yield misleading results. This is particularly true when data from experiments conducted under different conditions is combined; in this case, the differences in experimental setting, called *batch effects*, can throw off the analysis very dramatically. See Figure 1.4 .

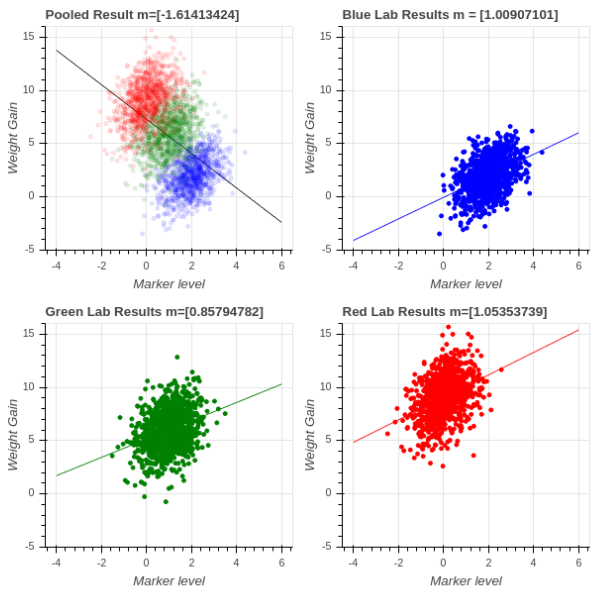


Figure 1.4: Simpson's Effect

1.7.3 Exercises

1. When proving that D is invertible if and only if the columns of X are linearly independent, we argued that if $X^\top Xm = 0$ for a nonzero vector m, then Xm is orthogonal to the span of the columns of X, and is also an element of that span, and is therefore zero. Provide the details: show that if H is a subspace of $\mathbf{R}^N$, and x is a vector in H such that $x \cdot h = 0$ for all $h \in H$, then $x = 0$.

2 Principal Component Analysis

2.1 Introduction

Suppose that, as usual, we begin with a collection of measurements of different features for a group of samples. Some of these measurements will tell us quite a bit about the difference among our samples, while others may contain relatively little information. For example, if we are analyzing the effect of a certain weight loss regimen on a group of people, the age and weight of the subjects may have a great deal of influence on how successful the regimen is, while their blood pressure might not. One way to help identify which features are more significant is to ask whether or not the feature varies a lot among the different samples. If nearly all the measurements of a feature are the same, it can't have much power in distinguishing the samples, while if the measurements vary a great deal then that feature has a chance to contain useful information.

In this section we will discuss a way to measure the variability of measurements and then introduce principal component analysis (PCA). PCA is a method for finding which linear combinations of measurements have the greatest variability and therefore might contain the most information. It also allows us to identify combinations of measurements that don't vary much at all. Combining this information, we can sometimes replace our original system of features with a smaller set that still captures most of the interesting information in our data, and thereby find hidden characteristics of the data and simplify our analysis a great deal.

2.2 Variance and Covariance

2.2.1 Variance

Suppose that we have a collection of measurements $(x_1, \ldots, x_n)$ of a particular feature X. For example, x_i might be the initial weight of the *ith* participant in our weight loss study. The mean of the values $(x_1, \ldots, x_n)$ is

$$\mu_X = \frac{1}{n}\sum_{i=1}^{n} x_i.$$

The simplest measure of the variability of the data is called its *variance.*

Definition: The (sample) variance of the data $x_1, \ldots, x_n$ is

$$\sigma_X^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu_X)^2 = \frac{1}{n}\left(\sum_{i=1}^{n} x_i^2\right) - \mu_X^2 \tag{2.1}$$

The square root of the variance is called the *standard deviation.*

As we see from the formula, the variance is a measure of how 'spread out' the data is from the mean.

Recall that in our discussion of linear regression we thought of our set of measurements $x_1, \ldots, x_n$ as a vector – it's one of the columns of our data matrix. From that point of view, the variance has a geometric interpretation – it is $\frac{1}{N}$ times the square of the distance from the point $X = (x_1, \ldots, x_n)$ to the point $\mu_X(1, 1, \ldots, 1) = \mu_X E$:

$$\sigma_X^2 = \frac{1}{n}(X - \mu_X E) \cdot (X - \mu_X E) = \frac{1}{n}\|X - \mu_X E\|^2. \tag{2.2}$$

2.2.2 Covariance

The variance measures the dispersion of measures of a single feature. Often, we have measurements of multiple features and we might want to know something about how two features are related. The *covariance* is a measure of whether two features tend to be related, in the sense that when one increases, the other one increases; or when one increases, the other one decreases.

Definition: Given measurements $(x_1, \ldots, x_n)$ and $(y_1, \ldots, y_n)$ of two features X and Y, the covariance of X and Y is

$$\sigma_{XY} = \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu_X)(y_i - \mu_Y) \tag{2.3}$$

There is a nice geometric interpretation of this, as well, in terms of the dot product. If $X = (x_1, \ldots, x_n)$ and $Y = (y_1 \ldots, y_n)$ then

$$\sigma_{XY} = \frac{1}{N}((X - \mu_X E) \cdot (Y - \mu_Y E)).$$

From this point of view, we can see that σ_{XY} is positive if the $X - \mu_X E$ and $Y - \mu_Y E$ vectors "point roughly in the same direction" and its negative if they "point roughly in the opposite direction."

2.2.3 Correlation

One problem with interpreting the variance and covariance is that we don't have a scale – for example, if σ_{XY} is large and positive, then we'd like to say that X and Y are closely related, but it could be just that the entries of $X-\mu_X E$ and $Y-\mu_Y E$ are large. Here, though, we can really take advantage of the geometric interpretation. Recall that the dot product of two vectors satisfies the formula

$$a\cdot b = \|a\|\|b\|\cos(\theta)$$

where θ is the angle between a and b. So

$$\cos(\theta) = \frac{a\cdot b}{\|a\|\|b\|}.$$

Let's apply this to the variance and covariance, by noticing that

$$\frac{(X-\mu_X E)\cdot(Y-\mu_Y E)}{\|(X-\mu_X E)\|\|(Y-\mu_Y E)\|} = \frac{\sigma_{XY}}{\sigma_{XX}\sigma_{YY}}$$

so the quantity

$$r_{XY} = \frac{\sigma_{XY}}{\sigma_X \sigma_Y} \tag{2.4}$$

measures the cosine of the angle between the vectors $X-\mu_X E$ and $Y-\mu_Y E$.

Definition: The quantity r_{XY} defined in Equation 2.4 is called the (sample) *correlation coefficient* between X and Y. We have $0\le |r_{XY}|\le 1$ with $r_{XY}=\pm 1$ if and only if the two vectors $X-\mu_X$ and $Y-\mu_Y$ are collinear in $\mathbf{R}^n$.

*Figure 2.1 illustrates data with different values of the correlation coefficient.

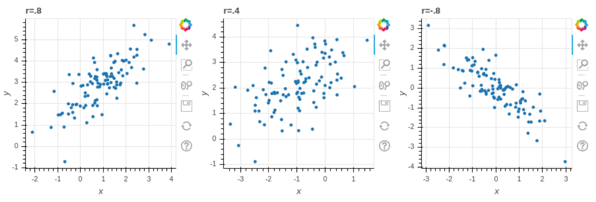


Figure 2.1: Correlation

2.2.4 The covariance matrix

In a typical situation we have many features for each of our (many) samples, that we organize into a data matrix X. To recall, each column of X corresponds to a feature that we measure, and each row corresponds to a sample. For example, each row of our matrix might correspond to a person enrolled in a study, and the columns correspond to height (cm), weight (kg), systolic blood pressure, and age (in years):

Table 2.1: A sample data matrix X

sample	Ht	Wgt	Bp	Age
A	180	75	110	35
B	193	80	130	40
...	...	...	...	...
U	150	92	105	55

If we have multiple features, as in this example, we might be interested in the variance of each feature and all of their mutual covariances. This "package" of information can be obtained "all at once" by taking advantage of some matrix algebra.

Definition: Let X be a $N \times k$ data matrix, where the k columns of X correspond to different features and the N rows to different samples. Let X_0 be the centered version of this data matrix, obtained by subtracting the mean μ_i of column i from all the entries x_{si} in that column. Then the $k \times k$ symmetric matrix

$$D_0 = \frac{1}{N} X_0^\top X_0$$

is called the (sample) covariance matrix for the data.

Proposition: The diagonal entries d_{ii} of D_0 are the variances of the columns of X:

$$d_{ii} = \sigma_i^2 = \frac{1}{N} \sum_{s=1}^{N} (x_{si} - \mu_i)^2$$

and the off-diagonal entries $d_{ij} = d_{ji}$ are the covariances of the i^{th} and j^{th} columns of X:

$$d_{ij} = \sigma_{ij} = \frac{1}{N} \sum_{s=1}^{N} (x_{si} - \mu_i)(x_{sj} - \mu_j)$$

The sum of the diagonal entries, the trace of D_0 is the **total** variance of the data.

Proof: This follows from the definitions, but it's worth checking the details, which we leave as an exercise.

2.2.5 Visualizing the covariance matrix

If the number of features in the data is not too large, a density matrix plot provides a tool for visualizing the covariance matrix of the data. A density matrix plot is an $k \times k$ grid of plots (where k is the number of features). The entry with (i, j) coordinates in the grid is a scatter plot of the i^{th} feature against the j^{th} one if $i \neq j$, and is a histogram of the i^{th} variable if $i = j$.

*Figure 2.2 is an example of a density matrix plot for a dataset with 50 samples and 2 features. This data has been centered, so it can be represented in a 50×2 data matrix X_0. The upper left and lower right graphs are scatter plots of the two columns, while the lower left and upper right are the histograms of the columns.

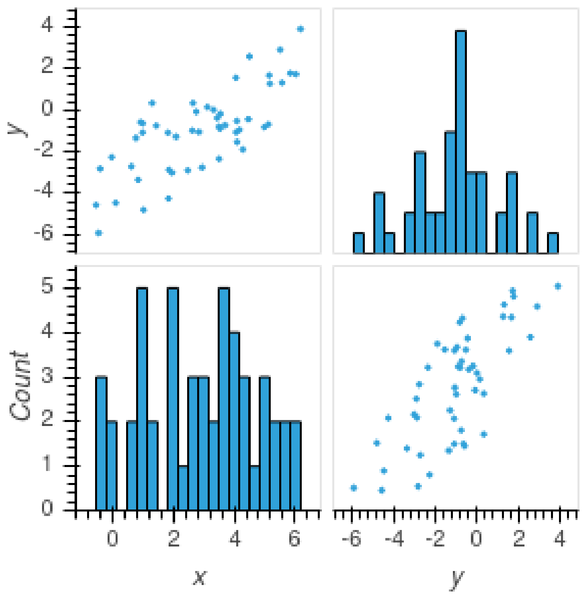


Figure 2.2: Density Matrix Plot

2.2.6 Linear Combinations of Features (Scores)

Sometimes useful information about our data can be revealed if we combine different measurements together to obtain a "hybrid" measure that captures something interesting. For example, in the Auto MPG dataset that we studied in the section on Linear Regression, we looked at the influence of both vehicle weight w and engine displacement e on gas mileage; perhaps their is some value in considering a hybrid "score" defined as

$$S = aw + be$$

for some constants a and b – maybe by choosing a good combination we could find a better predictor of gas mileage than using one or the other of the features individually.

As another example, suppose we are interested in the impact of the nutritional content of food on weight gain in a study. We know that both calorie content and the level dietary fiber contribute to the weight gain of participants eating this particular food; maybe there is some kind of combined "calorie/fiber" score we could introduce that captures the impact of that food better.

Definition: Let X_0 be a (centered) $N \times k$ data matrix giving information about k features for each of N samples. A linear synthetic feature, or a linear score, is a linear combination of the k features. The linear score is defined by constants $a_1, \dots, a_k$ so that If $y_1, \dots, y_k$ are the values of the features for a particular sample, then the linear score for that sample is

$$S = a_1 y_1 + a_2 y_2 + \cdots + a_k y_k$$

Lemma: The values of the linear score for each of the N samples can be calculated as

$$\begin{bmatrix} S_1 \\ \vdots \\ S_N \end{bmatrix} = X_0 \begin{bmatrix} a_1 \\ \vdots \\ a_k \end{bmatrix}. \tag{2.5}$$

Proof: Multiplying a matrix by a column vector computes a linear combination of the columns – that's what this lemma says. Exercise 3 asks you to write out the indices and make sure you believe this.

2.2.7 Mean and variance of scores

When we combine features to make a hybrid score, we assume that the features were centered to begin with, so that each features has mean zero. As a result, the mean of the hybrid features is again zero.

Lemma: A linear combination of features with mean zero again has mean zero.

Proof: Let S_i be the score for the i^{th} sample, so

$$S_i = \sum_{j=1}^{k} x_{ij} a_j.$$

where X_0 has entries x_{ij}. Then the mean value of the score is

$$\mu_S = \frac{1}{k}\sum_{i=1}^{N} S_i = \frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{k} x_{ij}a_j.$$

Reversing the order of the sum yields

$$\mu_S = \frac{1}{N}\sum_{j=1}^{k}\sum_{i=1}^{N} x_{ij}a_j = \sum_{j=1}^{k} a_j \frac{1}{N}(\sum_{i=1}^{N} x_{ij}) = \sum_{j=1}^{k} a_j\mu_j = 0$$

where $\mu_j = 0$ is the mean of the j^{th} feature (column) of X_0.

The variance is more interesting, and gives us an opportunity to put the covariance matrix to work. Remember from Equation 2.2 that, since a score S has mean zero, it's variance is $\sigma_S^2 = \frac{1}{N}S \cdot S$ – where here the score S is represented by the column vector with entries $S_1, \ldots S_k$ as in Equation 2.5.

Lemma: The variance of the score S with weights $a_1, \ldots a_k$ is

$$\sigma_S^2 = a^\top D_0 a = \begin{bmatrix} a_1 & \cdots & a_k \end{bmatrix} D_0 \begin{bmatrix} a_1 \\ \vdots \\ a_k \end{bmatrix} \tag{2.6}$$

More generally, if S_1 and S_2 are scores with weights $a_1, \ldots, a_k$ and $b_1, \ldots, b_k$ respectively, then the covariance $\sigma_{S_1 S_2}$ is

$$\sigma_{S_1S_2} = a^\top D_0 b.$$

Proof: From Equation 2.2 and Equation 2.5 we know that

$$\sigma_S^2 = \frac{1}{N} S \cdot S$$

and

$$S = X_0 a.$$

Since $\frac{1}{N}S \cdot S = \frac{1}{N}S^\top S$, this gives us

$$\frac{1}{N}\sigma_S^2 = \frac{1}{N}(X_0 a)^\top (X_0 a) = \frac{1}{N} a^\top X_0^\top X_0 a = a^\top D_0 a$$

as claimed.

For the covariance, use a similar argument with Equation 2.3 and Equation 2.5. writing $\sigma_{S_1S_2} = \frac{1}{N}S_1 \cdot S_2$ and the fact that S_1 and S_2 can be written as $X_0 a$ and $X_0 b$.

The point of this lemma is that the covariance matrix contains not just the variances and covariances of the original features, but also enough information to construct the variances and covariances for *any linear combination of features.*

In the next section we will see how to exploit this idea to reveal hidden structure in our data.

2.2.8 Geometry of Scores

Let's return to the dataset that we looked at in Section 2.2.5. We simplify the density matrix plot in Figure 2.3, which shows one of the scatter plots and the two histograms.

The scatter plot shows that the data points are arranged in a more or less elliptical cloud oriented at an angle to the xy-axes which represent the two given features. The two individual histograms show the distribution of the two features – each has mean zero, with the x-features distributed between -2 and 2 and the y feature between -4 and 4. Looking just at the two features individually, meaning only at the two histograms, we can't see the overall elliptical structure.

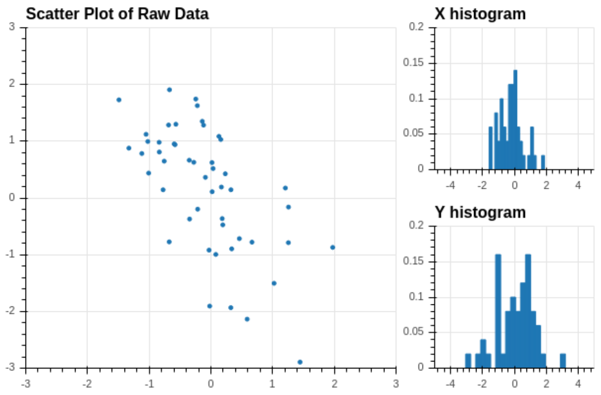


Figure 2.3: Simulated Data with Two Features

How can we get a better grip on our data in this situation? We can try to find a "direction" in our data that better illuminates the variation of the data. For example, suppose that we pick a unit vector at the origin pointing in a particular direction in our data. See Figure 2.4.

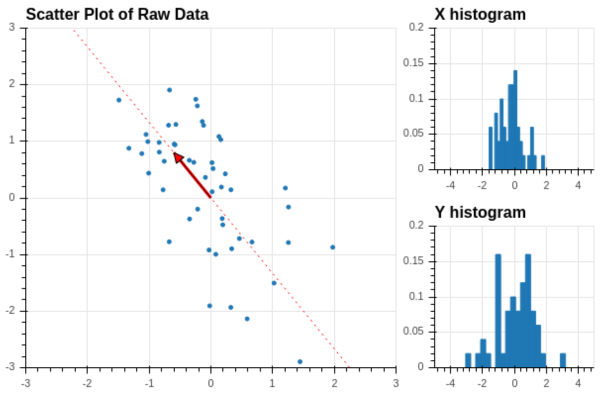


Figure 2.4: A direction in the data

Now we can orthogonally project the datapoints onto the line defined by this vector, as shown in Figure 2.5.

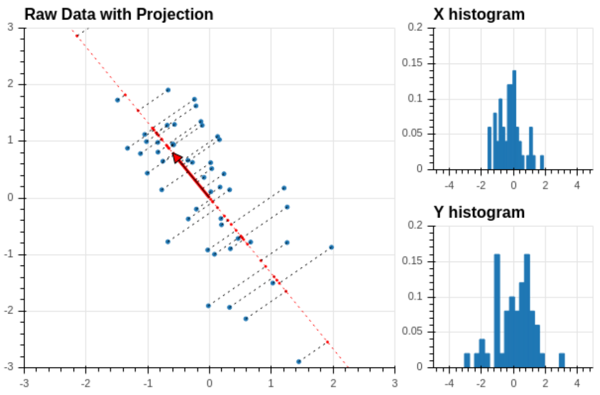


Figure 2.5: Projecting the datapoints

Recall that if the unit vector is defined by coordinates $u = [u_0, u_1]$, then the orthogonal projection of the point x with coordinates (x_0, x_1) is $(x \cdot u)u$. Now

$$x \cdot u = u_0 x_0 + u_1 x_1$$

so the coordinates of the points along the line defined by u are the values of the score Z defined by $u = [u_0, u_1]$. Using our work in the previous section, we see that we can find all of these coordinates by matrix multiplication:

$$Z = X_0 u$$

where X_0 is our data matrix. Now let's add a histogram of the values of Z to our picture:

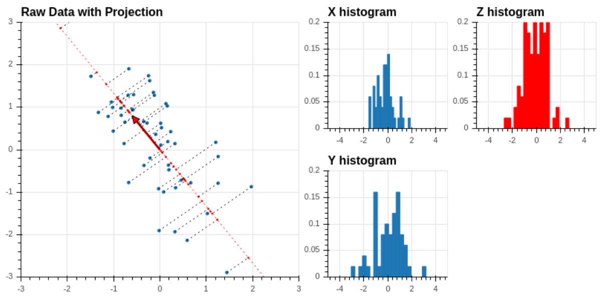


Figure 2.6: Distribution of Z

This histogram shows the distribution of the values of Z along the tilted line defined by the unit vector u.

Finally, using our work on the covariance matrix, we see that the variance of Z is given by

$$\sigma_Z^2 = \frac{1}{50} u^\top X_0^\top X_0 u = u^\top D_0 u$$

where D_0 is the covariance matrix of the data X_0.

Lemma: Let X_0 be a $N \times k$ centered data matrix, and let $D_0 = \frac{1}{N} X_0^\top X_0$ be the associated covariance matrix. Let u be a unit vector in "feature space" $\mathbf{R}^k$. Then the score $S = X_0 u$ can be interpreted as the coordinates of the points of X_0 projected onto the line generated by u. The variance of this score is

$$\sigma_S^2 = u^\top D_0 u = \sum_{i=1}^N s_i^2$$

where $s_i = X_0[i,:]u$ is the dot product of the i^{th} row $X_0[i,:]$ with u. It measures the variability in the data "in the direction of the unit vector u".

2.3 Principal Components

2.3.1 Change of variance with direction

As we've seen in the previous section, if we choose a unit vector u in the feature space and find the projection $X_0 u$ of our data onto the line through u, we get a "score" that we can use to measure the variance of the data in the direction of u. What happens as we vary u?

To study this question, let's continue with our simulated data from the previous section, and introduce a unit vector

$$u(\theta) = \begin{bmatrix} \cos(\theta) & \sin(\theta) \end{bmatrix}.$$

This is in fact a unit vector, since $\sin^2(\theta) + \cos^2(\theta) = 1$, and it is oriented at an angle θ from the x-axis.

The variance of the data in the direction of $u(\theta)$ is given by

$$\sigma_\theta^2 = u(\theta)^\top D_0 u(\theta).$$

A plot of this function for the data we have been considering is in Figure 2.7. As you can see, the variance goes through two full periods with the angle, and it reaches a maximum and minimum value at intervals of $\pi/2$ – so the two angles where the variance are maximum and minimum are orthogonal to one another.

The two directions where the variance is maximum and minimum are drawn on the original data scatter plot in Figure 2.8 .

Let's try to understand why this is happening.

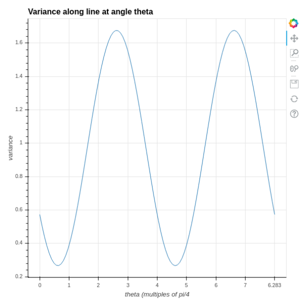


Figure 2.7: Change of variance with angle theta

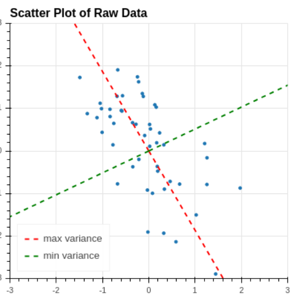


Figure 2.8: Data with principal directions

2.3.2 Directions of extremal variance

Given our centered, $N \times i$ data matrix X_0, with its associated covariance matrix $D_0 = \frac{1}{N} X_0^\top X_0$, we would like to find unit vectors u in $\mathbf{R}^k$ so that

$$\sigma_u^2 = u^\top D_0 u$$

reaches its maximum and its minimum. Here σ_u^2 is the variance of the "linear score" $X_0 u$ and it represents how dispersed the data is in the "u direction" in $\mathbf{R}^k$.

In this problem, remember that the coordinates of $u = (u_1, \ldots, u_k)$ are the variables and the symmetric matrix D_0 is given. As usual, we to find the maximum and minimum values of σ_u^2, we should look at the partial derivatives of σ_u^2 with respect to the variables u_i and set them to zero. Here, however, there is a catch – we want to restrict u to being a unit vector, with $u \cdot u = \sum u_i^2 = 1$.

So this is a *constrained optimization problem*:

- Find extreme values of the function

$$\sigma_u^2 = u^\top D_0 u$$

- Subject to the constraint $\|u\|^2 = u \cdot u = 1$ (or $u \cdot u - 1 = 0$)

We will use the technique of *Lagrange Multipliers* to solve such a problem.

To apply this method, we introduce the function

$$S(u, \lambda) = u^\top D_0 u - \lambda(u \cdot u - 1) \tag{2.7}$$

Then we compute the gradient

$$\nabla S = \begin{bmatrix} \frac{\partial S}{\partial u_1} \\ \vdots \\ \frac{\partial S}{\partial u_k} \\ \frac{\partial S}{\partial \lambda} \end{bmatrix} \tag{2.8}$$

and solve the system of equations $\nabla S = 0$. Here we have written the gradient as a column vector for reasons that will become clearer shortly.

Computing all of these partial derivatives looks messy, but actually if we take advantage of matrix algebra it's not too bad. The following two lemmas explain how to do this.

Lemma: Let M be a $N \times k$ matrix with constant coefficients and let u be a $k \times 1$ column vector whose entries are $u_1, \dots u_k$. The function $F(u) = Mu$ is a linear map from $\mathbf{R}^k \to \mathbf{R}^N$. Its (total) derivative is a linear map between the same vector spaces, and satisfies

$$D(F)(v) = Mv$$

for any $k \times 1$ vector v. If u is a $1 \times N$ matrix, and $G(u) = uM$, then

$$D(G)(v) = vM$$

for any $1 \times N$ vector v. (This is the matrix version of the derivative rule that $\frac{d}{dx}(ax) = a$ for a constant a.)

Proof: Since $F : \mathbf{R}^k \to \mathbf{R}^N$, we can write out F in more traditional function notation as

$$F(u) = (F_1(u_1, \dots, u_k), \dots, F_N(u_1, \dots, u_k)$$

where

$$F_i(u_1, \dots u_k) = \sum_{j=1}^{k} m_{ij} u_j.$$

Thus $\frac{\partial F_i}{\partial u_j} = m_{ij}$. The total derivative $D(F)$ is the linear map with matrix

$$D(F)_{ij} = \frac{\partial F_i}{\partial u_j} = m_{ij}$$

and so $D(F) = M$.

The other result is proved the same way.

Lemma: Let D be a symmetric $k \times k$ matrix with constant entries and let u be an $k \times 1$ column vector of variables $u_1, \dots, u_k$. Let $F : \mathbf{R}^k \to R$ be the function $F(u) = u^\top Du$. Then the gradient $\nabla_u F$ is a vector field – that is, a vector-valued function of u, and is given by the formula

$$\nabla_u F = 2Du$$

Proof: Let d_{ij} be the i, j entry of D. We can write out the function F to obtain

$$F(u_1, \dots, u_k) = \sum_{i=1}^{k} \sum_{j=1}^{k} u_i d_{ij} u_j.$$

Now $\frac{\partial F}{\partial u_i}$ is going to pick out only terms where u_i appears, yielding:

$$\frac{\partial F}{\partial u_i} = \sum_{j=1}^{k} d_{ij} u_j + \sum_{j=1}^{k} u_j d_{ji}$$

Here the first sum catches all of the terms where the first "u" is u_i; and the second sum catches all the terms where the second "u" is u_i. The diagonal terms $u_i^2 d_{ii}$ contribute once to each sum, which is consistent with the rule that the derivative of $u_i^2 d_{ii} = 2u_i d_{ii}$. To finish the proof, notice that

$$\sum_{j=1}^{k} u_j d_{ji} = \sum_{j=1}^{k} d_{ij} u_j$$

since D is symmetric, so in fact the two terms are the same Thus

$$\frac{\partial}{\partial u_i} F = 2 \sum_{j=1}^{k} d_{ij} u_j$$

But the right hand side of this equation is twice the i^{th} entry of Du, so putting the results together we get

$$\nabla_u F = \begin{bmatrix} \frac{\partial F}{\partial u_1} \\ \vdots \\ \frac{\partial F}{\partial u_k} \end{bmatrix} = 2Du.$$

The following theorem puts all of this work together to reduce our questions about how variance changes with direction.

2.3.3 Critical values of the variance

Theorem: The critical values of the variance σ_u^2, as u varies over unit vectors in $\mathbf{R}^N$, are the eigenvalues $\lambda_1, \ldots, \lambda_k$ of the covariance matrix D, and if e_i is a unit eigenvector corresponding to λ_i, then $\sigma_{e_i}^2 = \lambda_i$.

Proof: Recall that we introduced the Lagrange function $S(u, \lambda)$, whose critical points give us the solutions to our constrained optimization problem. As we said in Equation 2.7:

$$S(u, \lambda) = u^\top D_0 u - \lambda(u \cdot u - 1) = u^\top D_0 u - \lambda(u \cdot u) + \lambda$$

Now apply our Matrix calculus lemmas. First, let's treat λ as a constant and focus on the u variables. We can write $u \cdot u = u^\top I_N u$ where I_N is the identity matrix to compute:

$$\nabla_u S = 2D_0 u - 2\lambda u$$

For λ we have

$$\frac{\partial}{\partial \lambda} S = -u \cdot u + 1.$$

The critical points occur when

$$\nabla_u S = 2(D_0 - \lambda)u = 0$$

and

$$\frac{\partial}{\partial \lambda} S = 1 - u \cdot u = 0$$

The first equation says that λ must be an eigenvalue, and u an eigenvector:

$$D_0 u = \lambda u$$

while the second says u must be a unit vector $u \cdot u = \|u\|^2 = 1$. The second part of the result follows from the fact that if e_i is a unit eigenvector with eigenvalue λ_i then

$$\sigma_{e_i}^2 = e_i^\top D_0 e_i = \lambda_i \|e_i\|^2 = \lambda_i.$$

To really make this result pay off, we need to recall some key facts about the eigenvalues and eigenvectors of symmetric matrices. Because these facts are so central to this result, and to other applications throughout machine learning and mathematics generally, we provide proofs in Section 2.5.

Table 2.2: Properties of Eigenvalues of Real Symmetric Matrices

Summary
1. All of the eigenvalues $\lambda_1, \dots, \lambda_l$ of D are real. If $u^\top Du \geq 0$ for all $u \in \mathbf{R}^k$, then all eigenvalues λ_i are non-negative. In the latter case we say that D is *positive semi-definite.*   2. If v is an eigenvector for D with eigenvalue λ, and w is an eigenvector with a different eigenvalue λ', then v and w are orthogonal: $v \cdot w = 0$.   3. There is an orthonormal basis $u_1, \dots, u_k$ of $\mathbf{R}^k$ made up of eigenvectors of D corresponding to the eigenvalues λ_i.   4. Let Λ be the diagonal matrix with entries $\lambda_1, \dots, \lambda_N$ and let P be the matrix whose columns are made up of the vectors u_i. Then $D = P\Lambda P^\top$.

If we combine our theorem on the critical values with the spectral theorem we get a complete picture. Let D_0 be the covariance matrix of our data. Since

$$\sigma_u^2 = u^\top D_0 u \geq 0 (\text{it's a sum of squares})$$

we know that the eigenvalues $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_k \geq 0$ are all nonnegative. Choose a corresponding sequence $u_1, \dots u_k$ of orthogonal eigenvectors where all $\|u_i\|^2 = 1$. Since the u_i form a basis of $\mathbf{R}^N$, any score is a linear combination of the u_i:

$$S = \sum_{i=1}^{k} a_i u_i.$$

Since $u_i^\top D_0 u_j = \lambda_j u_i^\top u_j = 0$ unless $i = j$, in which case it is λ_i, we can compute

$$\sigma_S^2 = \sum_{i=1}^{k} \lambda_i a_i^2,$$

and $\|S\|^2 = \sum_{i=1}^{k} a_i^2$ since the u_i are an orthonormal set. So in these coordinates, our optimization problem is:

- maximize $\sum \lambda_i a_i^2$
- subject to the constraint $\sum a_i^2 = 1$.

We don't need any fancy math to see that the maximum happens when $a_1 = 1$ and the other $a_j = 0$, and in that case, the maximum is λ_1. (If λ_1 occurs more than once, there may be a whole subspace of directions where the variance is maximal). Similarly, the minimum value is λ_k and occurs when $a_k = 1$ and the others are zero.

2.3.4 Subspaces of extremal variance

We can generalize the idea of the variance of our data in a particular direction to a higher dimensional version of *total variance* in a subspace. Suppose that E is a subspace of $\mathbf{R}^k$ and U is a matrix whose columns span E – the columns of U are the weights of a family of scores that span E. The values of these scores are XU and the covariance matrix of this projected data is

$$\frac{1}{N}U^{\top}X^{\top}XU = U^{\top}D_0U.$$

.

Finally, the *total variance* σ_E^2 of the data projected into E is the sum of the diagonal entries of the matrix

$$\sigma_E^2 = trace(U^{\top}D_0U)$$

Just as the variance in a given direction u depends on the scaling of u, the variance in a subspace depends on the scaling of the columns of U. To normalize this scaling, we assume that the columns of U are an orthonormal basis of the subspace E.

Now we can generalize the question asked in Section 2.3.2 by seeking, not just a vector u pointing in the direction of the extremal variance, but instead the *subspace* U_s of dimension s with the property that the total variance of the projection of the data into U_s is maximal compared to its projection into other subspaces of that dimension. This is called a *subspace of extremal variance.*

To make this concrete, suppose we consider a subspace E of $\mathbf{R}^k$ of dimension t with basis $w_1, \ldots, w_t$. Complete this to a basis $w_1, \ldots, w_t, w_{t+1}, \ldots, w_k$ of $\mathbf{R}^k$ and then apply the Gram Schmidt Process (see Section 2.5.1) to find an orthonormal basis $w'_1, \ldots, w'_s, w'_{s+1}, \ldots, w'_k$ where the $w'_1, \ldots, w'_t$ are an orthonormal basis for E. Let W be the $k \times t$ matrix whose columns are the w'_i for $i = 1, \ldots, t$. The rows of the matrix X_0W given the coordinates of the projection of each sample into the subspace E expressed in terms of the scores corresponding to these vectors w'_i. The total variance of these projections is

$$\sigma_E^2 = \sum_{i=1}^{t} \|X_0w'_i\|^2 = \sum_{i=1}^{t}(w'_i)^{\top}X_0^{\top}X_0w'_i = \sum_{i=1}^{t}(w'_i)^{\top}D_0w'_i$$

If we want to maximize this, we have the constrained optimization problem of finding $w'_1, \ldots, w'_t$ so that

- $\sum_{i=1}^{t}(w'_i)^{\top}D_0w'_i$ is maximal
- subject to the constraint that each w_i has $\|w'_i\|^2 = 1$,
- and that the w'_i are orthogonal, meaning $w'_i \cdot w'_j = 0$ for $i \neq j$,
- and that the w'_i are linearly independent.

Then the span E of these w'_i is subspace of extremal variance.

Theorem: A t-dimensional subspace E is a subspace of extremal variance if and only if it is spanned by t orthonormal eigenvectors of the matrix D_0 corresponding to the t largest eigenvalues for D_0.

Proof: We can approach this problem using Lagrange multipliers and matrix calculus if we are careful. Our unknown is $k \times t$ matrix W whose columns are the t (unknown) vectors w'_i. The objective function that we are seeking to maximize is

$$F = trace(W^\top D_0 W) = \sum_{i=1}^{t} (w'_i)^\top D_0 w_i.$$

The constraints are the requirements that $\|w'_i\|^2 = 1$ and $w'_i \cdot w'_j = 0$ if $i \neq j$. If we introduction a matrix of lagrange multipliers $\Lambda = (\lambda_{ij})$, where λ_{ij} is the multiplier that goes with the the first of these constraints when $i = j$, and the second when $i \neq j$, we can express our Lagrange function as:

$$S(W, \Lambda) = trace(W^\top D_0 W) - (W^\top W - I)\Lambda$$

where I is the $t \times t$ identity matrix.

Taking the derivatives with respect to the entries of W and of Λ yields the following two equations:

$$\begin{aligned} D_0 W &= W\Lambda \\ W^\top W &= I \end{aligned}$$

The first of these equations says that the space E spanned by the columns of W is *invariant* under D_0, while the second says that the columns of W form an orthonormal basis.

Let's assume for the moment that we have a matrix W that satisfies these conditions.
Then it must be the case that Λ is a symmetric, real valued $t \times t$ matrix, since

$$W^\top D_0 W = W^\top W \Lambda = \Lambda.$$

and the matrix on the left is symmetric.

By the properties of real symmetric matrices (the spectral theorem), there are orthonormal vectors $q_1, \ldots q_t$ that are eigenvectors of Λ with corresponding eigenvalues τ_i. If we let Q be the matrix whose columns are the vectors q_i and let T be the diagonal $t \times t$ matrix whose entries are the τ_i, we have

$$\Lambda Q = QT.$$

If we go back to our original equations, we see that if W exists such that $DW = W\Lambda$, then there is a matrix Q with orthonormal columns and a diagonal matrix T such that

$$D_0 W Q = W \Lambda Q = W Q T.$$

In other words, WQ is a matrix whose columns are eigenvectors of D_0 with eigenvalues τ_i for $i = 1, \dots, t$.

Thus we see how to construct an invariant subspace E and a solution matrix W. Such an E is spanned by t orthonormal eigenvectors q_i with eigenvalues τ_i of D_0; and W is is the matrix whose columns are the q_i. Further, in that case, the total variance associated to E is the sum of the eigenvalues τ_i; to make this as large as possible, we should choose our eigenvectors to correspond to t of the largest eigenvalues of D_0. This concludes the proof.

2.3.5 Definition of Principal Components

Definition: The orthonormal unit eigenvectors u_i for D_0 are the *principal directions* or *principal components* for the data X_0.

Theorem: The maximum variance occurs in the principal direction(s) associated to the largest eigenvalue, and the minimum variance in the principal direction(s) associated with the smallest one. The covariance between scores in principal directions associatedwith different eigenvalues is zero.

At this point, the picture in Figure 2.8 makes sense – the red and green dashed lines are the principal directions, they are orthogonal to one another, and the point in the directions where the data is most (and least) “spread out.”

Proof: The statement about the largest and smallest eigenvalues is proved at the very end of the last section. The covariance of two scores corresponding to different eigenvectors u_i and u_j is

$$u_i^\top D_0 u_j = \lambda_j (u_i \cdot u_j) = 0$$

since the u_i and u_j are orthogonal.

Sometimes the results above are presented in a slightly different form, and may be referred to, in part, as Rayleigh’s theorem.

Corollary: (Rayleigh’s Theorem) Let D be a real symmetric matrix and let

$$H(v) = \max_{v \neq 0} \frac{v^\top D v}{v^\top v}.$$

Then $H(v)$ is the largest eigenvalue of D. (Similarly, if we replace max by min, then the minimum is the least eigenvalue).

Proof: The maximum of the function $H(v)$ is the solution to the same optimization problem that we considered above.

Exercises.

1. Prove that the two expressions for σ_X^2 given in Equation 2.1 are the same.

2. Prove that the covariance matrix is as described in the proposition in Section 2.2.4.
3. Let X_0 be a $k \times N$ matrix with entries x_{ij} for $1 \leq i \leq k$ and $1 \leq j \leq N$. If a linear score is defined by the constants $a_1, \ldots a_N$, check that equation Equation 2.5 holds as claimed.
4. Why is it important to use a unit vector when computing the variance of X_0 in the direction of u? Suppose $v = \lambda u$ where u is a unit vector and $\lambda > 0$ is a constant. Let S' be the score $X_0 v$. How is the variance of S' related to that of $S = X_0 u$?

2.4 Dimensionality Reduction via Principal Components

The principal components associated with a dataset separate out directions in the feature space in which the data is most (or least) variable. One of the main applications of this information is to enable us to take data with a great many features – a set of points in a high dimensional space – and, by focusing our attention on the scores corresponding to the principal directions, capture most of the information in the data in a much lower dimensional setting.

To illustrate how this is done, let X be a $N \times k$ data matrix, let X_0 be its centered version, and let $D_0 = \frac{1}{N} X_0^\top X$ be the associated covariance matrix.

Apply the spectral theorem (proved in Section 2.5) to the covariance matrix to obtain eigenvalues $\lambda_1 \geq \lambda_2 \geq \cdots \lambda_k \geq 0$ and associated eigenvectors $u_1, \ldots, u_k$. The scores $S_i = X_0 u_i$ give the values of the data in the principal directions. The variance of S_i is λ_i.

Now choose a number $t < k$ and consider the vectors $S_1, \ldots, S_t$. The j^{th} entry in S_i is the value of the score S_i for the j^{th} data point. Because $S_1, \ldots, S_t$ capture the most significant variability in the original data, we can learn a lot about our data by considering just these t features of the data, instead of needing all N.

To illustrate, let's look at an example. We begin with a synthetic dataset X_0 which has 200 samples and 15 features. The data (some of it) for some of the samples is shown in Table 2.3.

Table 2.3: Simulated Data for PCA Analysis

	f-0	f-1	f-2	f-3	f-4	...	f-10	f-11	f-12	f-13	f-14
s-0	1.18	-0.41	2.02	0.44	2.24	...	0.32	0.95	0.88	1.10	0.89
s-1	0.74	0.58	1.54	0.23	2.05	...	0.99	1.14	1.56	0.99	0.59
...	...	...	...	...	...	...	...	...	...	...	...
s-198	1.04	2.02	1.44	0.40	1.33	...	0.62	0.62	0.54	1.96	0.04
s-199	0.92	2.09	1.58	1.19	1.17	...	0.42	0.85	0.83	2.22	0.90

The full dataset is a 200×15 matrix; it has 3000 numbers in it and we're not really equipped to make sense of it. We could try some graphing – for example, Figure 2.9 shows a scatter plot of two of the features plotted against each other.

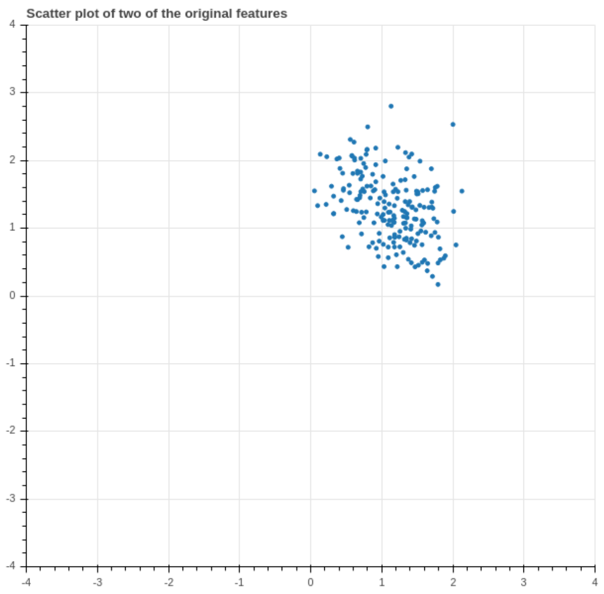


Figure 2.9: Scatter Plot of Two Features

Unfortunately there's not much to see in Figure 2.9 – just a blob – because the individual features of the data don't tell us much in isolation, whatever structure there is in this data arises out of the relationship between different features.

In Figure 2.10 we show a "density grid" plot of the data. The graph in position i, j shows a scatter plot of the i^{th} and j^{th} columns of the data, except in the diagonal positions, where in position i, i we plot a histogram of column i. There's not much structure visible; it is a lot of blobs.

So let's apply the theory of principal components. We use a software package to compute the eigenvalues and eigenvectors of the matrix D_0. The 15 eigenvalues $\lambda_1 \geq \cdots \geq \lambda_{15}$ are plotted, in descending order, in Figure 2.11 .

This plot shows that the first 4 eigenvalues are relatively large, while the remaining 11 are smaller and not much different from each other. We interpret this as saying that *most of the variation in the data is accounted for by the first four principal components.* We can even make this quantitative. The *total variance* of the data is the sum of the eigenvalues of the covariance matrix – the trace of D_0 – and in this example that sum is around 5. The sum of the first 4 eigenvalues is about 4, so the first four eignvalues account for about 4/5 of the total variance, or about 80% of the variation of the data.

Now let's focus in on the two largest eigenvalues λ_1 and λ_2 and their corresponding eigenvectors u_1 and u_2. The 200×1 column vectors $S_1 = X_0 u_1$ and $S_2 = X_0 u_2$ are the values of the scores associated with these two eigenvectors. So for each data point (each row of X_0) we have two

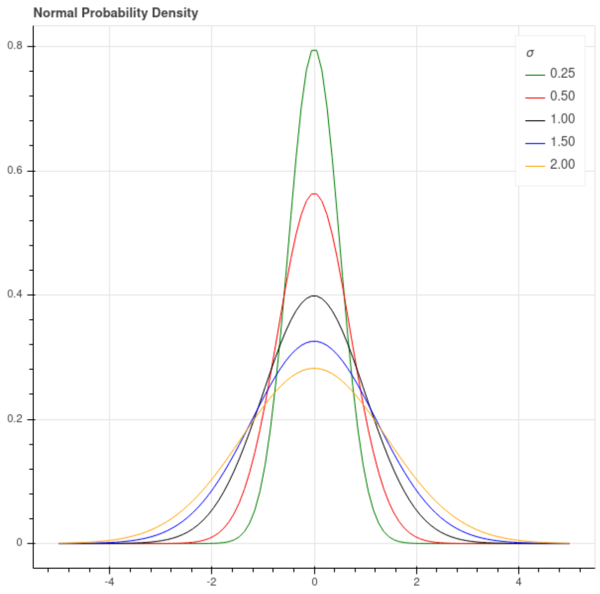


Figure 2.10: Density Grid Plot of All Features

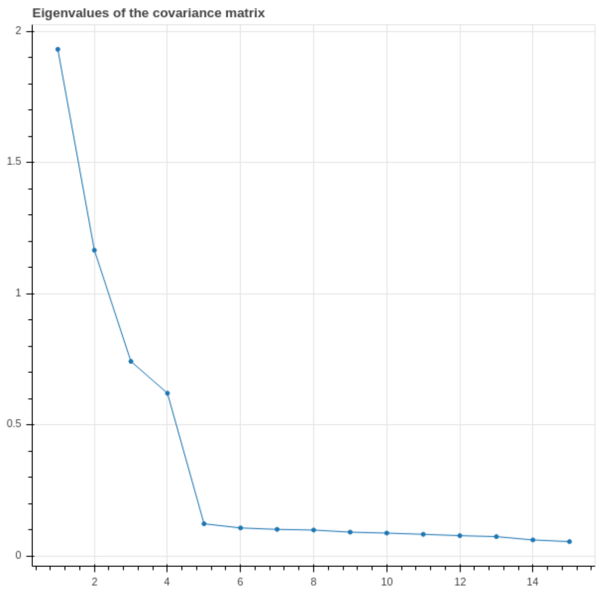


Figure 2.11: Eigenvalues of the Covariance Matrix

values (the corresponding entries of S_1 and S_2.) In Figure 2.12 we show a scatter plot of these scores.

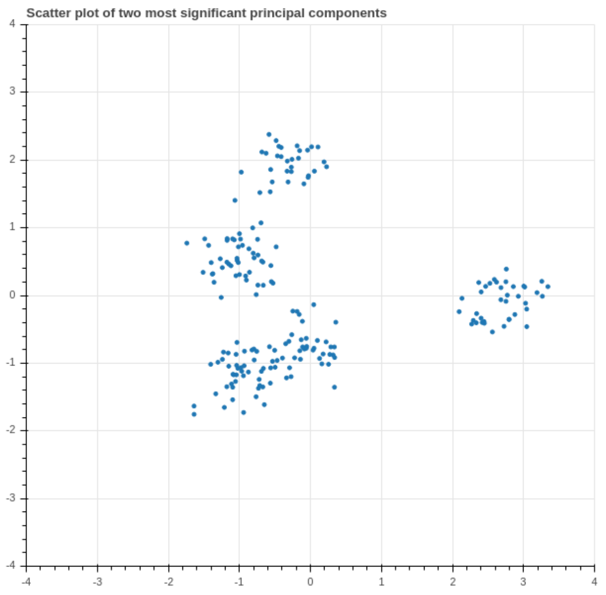


Figure 2.12: Scatter Plot of Scores in the First Two Principal Directions

Notice that suddenly some structure emerges in our data! We can see that the 200 points are separated into five clusters, distinguished by the values of their scores! This ability to find hidden structure in complicated data, is one of the most important applications of principal components.

If we were dealing with real data, we would now want to investigate the different groups of points to see if we can understand what characteristics the principal components have identified.

2.4.1 Loadings

There's one last piece of the PCA puzzle that we are going to investigate. In Figure 2.12, we plotted our data points in the coordinates given by the first two principal components. In geometric terms, we took the cloud of 200 points in $\mathbf{R}^{15}$ given by the rows of X_0 and projected those points into the two dimensional plane spanned by the eigenvectors u_1 and u_2, and then plotted the distribution of the points in that plane.

More generally, suppose we take our dataset X_0 and consider the first t principal components corresponding to the eigenvectors $u_1, \ldots, u_t$. The projection of the data into the space spanned by these eigenvectors is the represented by the $S = k \times t$ matrix $X_0 U$ where U is the $k \times t$

matrix whose columns are the eigenvectors u_i. Each row of S gives the values of the score arising from u_i in the i^{th} column for $i = 1, \ldots, t$.

The remaining question that we wish to consider is: how can we see some evidence of the original features in subspace? We can answer this by imagining that we had an artificial sample x that has a measurement of 1 for the i^{th} feature and a measurement of zero for all the other features. The corresponding point is represented by a $1 \times k$ row vector with a 1 in position i. The projection of this synthetic sample into the span of the first t principal components is the $1 \times t$ vector xU. Notice, however, that xU is just the i^{th} row of the matrix U. This vector in the space spanned by the u_i is called the "loading" of the i^{th} feature in the principal components.

This is illustrated in Figure 2.13, which shows a line along the direction of the loading corresponding to the each feature added to the scatter plot of the data in the plane spanned by the first two principal components. One observation one can make is that some of the features are more "left to right", like features 7 and 8, while others are more "top to bottom", like 6. So points that lie on the left side of the plot have smaller values of features 7 and 8, while those at the top of the plot have larger values of feature 6.

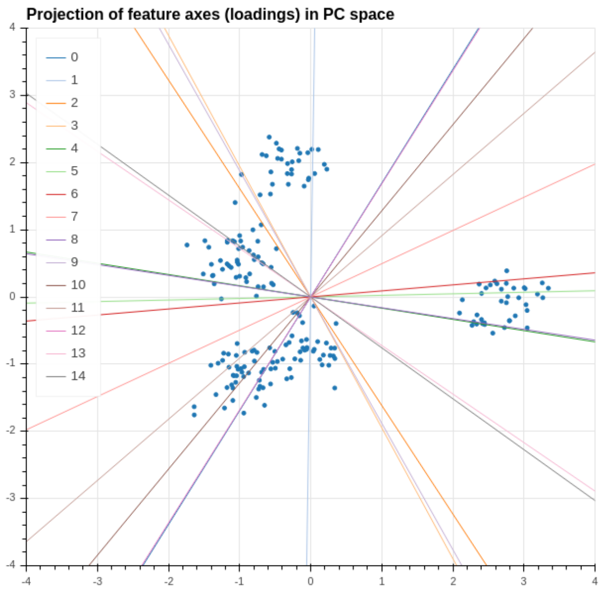


Figure 2.13: Loadings in the Principal Component Plane

2.4.2 The singular value decomposition

The singular value decomposition is a slightly more detailed way of looking at principal components. Let Λ be the diagonal matrix of eigenvalues of D_0 and let P be the $k \times k$ orthogonal matrix whose columns are the principal components. Then we have

$$D_0 = \frac{1}{N}X_0^\top X_0 = P\Lambda P^\top.$$

Consider the $N \times k$ matrix

$$X_0 P = A.$$

As we saw in the previous section, the columns of A give the projection of the data into the k principal directions. Then

$$A^\top A = P^\top X_0^\top X_0 P = N\Lambda.$$

In other words, the columns of A are orthogonal and the diagonal entries of $A^\top A$ are N times the variance of the data in the various principal directions.

Now we are going to tinker with the matrix A in order to make an $N \times N$ orthogonal matrix. The first modification we make is to normalize the columns of A so that they have length 1. We do this by setting

$$A_1 = A(N\Lambda)^{-1/2}.$$

Then $A_1^\top A_1$ is the identity, so the columns of A_1 are orthonormal. Here we are assuming that the eigenvalues of D_0 are nonzero – this isn't strictly necessary, and we could work around this, but for simplicity we will assume it's true. It amounts to the assumption that the independent variables are not linearly related, as we've seen before.

The second modification is to extend A_1 to an $N \times N$ matrix. The k columns of A_1 span only a k-dimensional subspace of the N-dimensional space where the feature vectors lie.
Complete the subspace by finding an orthogonal complement to it – that is, find $N-k$ mutually orthogonal unit vectors all orthogonal to the column space of A_1. By adding these vectors to A as columns, create an extended $N \times N$ matrix $\tilde{A}_1$ which is orthogonal.

Notice that $\tilde{A}_1^\top A$ is an $N \times k$ matrix whose upper $k \times k$ block is $(N\Lambda)^{1/2}$ and whose final $N-k$ rows are all zero. We call this matrix $\tilde{\Lambda}$.

To maintain consistency with the traditional formulation, we let $U = \tilde{A}_1^\top$ and then we have the following proposition.

Proposition: We have a factorization

$$X_0 = U\tilde{\Lambda}P^\top \tag{2.9}$$

where U and P are orthogonal matrices of size $N \times N$ and $k \times k$ respectively, and $\tilde{\Lambda}$ is an $N \times k$ diagonal matrix. This is called the "singular value decomposition" of X_0, and the entries of $\tilde{\Lambda}$ are called the singular values. If we let $u_1, \ldots, u_k$ be the first k rows of U, then the k column vectors $u_i^\top$ are an orthonormal basis for the feature space spanned by the columns of X_0, and they point in the "principal directions" for the data matrix X_0.

In this section we take a slight detour and apply what we've learned about the covariance matrix, principal components, and the singular value decomposition to the original problem of linear regression that we studied in Chapter 1.

In this setting, in addition to our centered data matrix X_0, we have a vector Y of target values and we find the "best" approximation

$$\hat{Y} = X_0 M$$

using the least squares method. As we showed in Chapter 1, the optimum M is found as

$$M = (X_0^\top X_0)^{-1} X^\top Y = N D_0^{-1} X_0^\top Y$$

and the predicted values $\hat{Y}$ are

$$\hat{Y} = N X_0 D_0^{-1} X_0^\top Y.$$

Geometrically, we understood this process as defining $\hat{Y}$ to be the orthogonal projection of Y into the subspace spanned by the columns of X_0.

Let's use the decomposition (see Equation 2.9) $X_0 = U\tilde{\Lambda}P^\top$ in this formula. First, notice that

$$X_0^\top X_0 = P\tilde{\Lambda}^\top U^\top U \tilde{\Lambda} P^\top = P\tilde{\Lambda}^\top \tilde{\Lambda} P^\top.$$

The middle term $\tilde{\Lambda}^\top\tilde{\Lambda}$ is the $k \times k$ matrix Λ whose diagonal entries are $N\lambda_i$ where λ_i are the eigenvalues of the covariance matrix D_0. Assuming these are all nonzero (which is tantamount to the assumption that the covariance matrix is invertible), we obtain

$$\hat{Y} = NU\tilde{\Lambda}P^\top P \Lambda^{-1} P^\top P \tilde{\Lambda}^\top U^\top Y.$$

There is a lot of cancellation here, and in the end what's left is

$$\hat{Y} = UEU^\top Y$$

where E is and $N \times N$ matrix whose upper $k \times k$ block is the identity and whose remaining entries are zero. Rearranging a bit more we have

$$U^\top \hat{Y} = EU^\top Y.$$

To unpack this equation, let $u_1, \ldots, u_N$ be the rows of the matrix U. Since U is an orthogonal matrix, the column vectors $u_i^\top$ are an orthonormal basis for the N dimensional space where the columns of X_0 lie. We can write the target vector Y

$$Y = \sum_{j=1}^{N} (u_j \cdot Y) u_j^\top.$$

Then the projection $\hat{Y}$ of Y into the subspace spanned by the data is obtained by dropping the last $N-k$ terms in the sum:

$$\hat{Y} = \sum_{j=1}^{k} (u_j \cdot Y) u_j^\top$$

2.5 Eigenvalues and Eigenvectors of Real Symmetric Matrices (The Spectral Theorem)

Now that we've shown how to apply the theory of eigenvalues and eigenvectors of symmetric matrices to extract principal directions from data, and to use those principal directions to find structure, we will give a proof of the properties that we summarized in Table 2.2.

A key tool in the proof is the Gram-Schmidt orthogonalization process.

2.5.1 Gram-Schmidt

Proposition (Gram-Schmidt Process): Let $w_1, \ldots, w_k$ be a collection of linearly independent vectors in $\mathbf{R}^N$ and let W be the span of the w_i. Let $u_1 = w_1$ and let

$$u_i = w_i - \sum_{j=1}^{i-1} \frac{w_i \cdot u_j}{u_j \cdot u_j} u_j$$

for $i = 2, \ldots, k$. Then

- The vectors u_i are orthogonal: $u_i \cdot u_j = 0$ unless $i = j$.
- The vectors u_i span W.
- Each u_i is orthogonal to the all of $w_1, \ldots, w_{i-1}$.
- The vectors $u_i' = u_i / \|u_i\|$ are orthonormal.

Proof: This is an inductive exercise, and we leave it to you to work out the details.

2.5.2 The spectral theorem

Theorem: Let D be a real symmetric $N \times N$ matrix. Then:

1. All of the N eigenvalues $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_N$ are real. If $u^\top D u \geq 0$ for all $u \in \mathbf{R}^N$, then all eigenvalues $\lambda_i \geq 0$.
2. The matrix D is diagonalizable – that is, it has N linearly independent eigenvectors.
3. If v and w are eigenvectors corresponding to eigenvalues λ and λ', with $\lambda \neq \lambda'$, then v and w are orthogonal: $v \cdot w = 0$.

4. There is an orthonormal basis $u_1, \dots, u_N$ of $\mathbf{R}^N$ made up of eigenvectors for the eigenvalues λ_i.
5. Let Λ be the diagonal matrix with entries $\lambda_1, \dots, \lambda_N$ and let P be the matrix whose columns are made up of the eigenvectors u_i. Then $D = P\Lambda P^\top$.

Proof: Start with 1. Suppose that λ is an eigenvalue of D. Let u be a corresponding nonzero eigenvector. Then $Du = \lambda u$ and $D\overline{u} = \overline{\lambda}\overline{u}$, where $\overline{u}$ is the vector whose entries are the conjugates of the entries of u (and $\overline{D} = D$ since D is real). Now we have

$$\overline{u}^\top Du = \lambda\overline{u} \cdot u = \lambda\|u\|^2$$

and

$$u^\top D\overline{u} = \overline{\lambda}u \cdot \overline{u} = \overline{\lambda}\|u\|^2.$$

But the left hand side of both of these equations are the same (take the transpose and use the symmetry of D) so we must have $\lambda\|u\|^2 = \overline{\lambda}\|u\|^2$ so $\lambda = \overline{\lambda}$, meaning λ is real.

If we have the additional property that $u^\top Du \geq 0$ for all u, then in particular $u_i^\top Du_i = \lambda\|u\|^2 \geq 0$, and since $\|u\|^2 > 0$ we must have $\lambda \geq 0$.

Property 2 is in some ways the most critical fact. We know from the general theory of the characteristic polynomial, and the fundamental theorem of algebra, that D has N complex eigenvalues, although some may be repeated. However, it may not be the case that D has N linearly independent eigenvectors – it may not be *diagonalizable.* So we will establish that now.

A one-by-one matrix is automatically symmetric and diagonalizable. In the N-dimensional case, we know, at least, that D has at least one eigenvector, and real one at that by part 1, and this gives us a place to begin an inductive argument.

Let $v_N \neq 0$ be an eigenvector with eigenvalue λ and normalized so that $\|v_N\|^2 = 1$,
and extend this to a basis $v_1, \dots v_N$ of $\mathbf{R}^N$. Apply the Gram-Schmidt process to construct an orthonormal basis of $\mathbf{R}^N$ $u_1, \dots, u_N$ so that $u_N = v_N$.

Any vector $v \in \mathbf{R}^N$ is a linear combination

$$v = \sum_{i=1}^{N} a_i u_i$$

and, since the u_i are orthonormal, the coefficients can be calculated as $a_i = (u_i \cdot v)$.

Using this, we can find the matrix D' of the linear map defined by our original matrix D in this new basis. By definition, if d'_{ij} are the entries of D', then

$$Du_i = \sum_{j=1}^{N} d'_{ij} u_j$$

and so

$$d'_{ij} = u_j \cdot Du_i = u_j^\top Du_i.$$

Since D is symmetric, $u_j^\top Du_i = u_i^\top Du_j$ and so $d'_{ij} = d'_{ji}$. In other words, the matrix D' is still symmetric. Furthermore,

$$d'_{Ni} = u_i \cdot Du_N = u_i \cdot \lambda u_N = \lambda(u_i \cdot u_N)$$

since $u_N = v_N$. Since the u_i are an orthonormal basis, we see that $d'_{iN} = 0$ unless $i = N$, and $d'_{NN} = \lambda$.

In other words, the matrix D' has a block form:

$$D' = \begin{pmatrix} * & * & \cdots & * & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ * & * & \cdots & * & 0 \\ 0 & 0 & \cdots & 0 & \lambda \end{pmatrix}$$

and the block denoted by $*$'s is symmetric. If we call that block D_*, the inductive hypothesis tells us that the symmetric matrix D_* is diagonalizable, so it has a basis of eigenvectors $u'_1, \ldots, u'_{N-1}$ with eigenvalues $\lambda_1, \ldots, \lambda_{N-1}$; this gives us a basis for the subspace of $\mathbf{R}^N$ spanned by $u_1, \ldots, u_{N-1}$ which, together with u_N gives us a basis of $\mathbf{R}^N$ consisting of eigenvectors of D.

This finishes the proof of Property 2.

For property 3, compute

$$v^\top Dw = \lambda'(v \cdot w) = w^\top Dv = \lambda(w \cdot v).$$

Since $\lambda \neq \lambda'$, we must have $v \cdot w = 0$.

For property 4, if the eigenvalues are all distinct, this is a consequence of property 2 – you have N eigenvectors, scaled to length 1, for different eigenvalues, and by 2 they are orthogonal. So the only complication is the case where some eigenvalues are repeated. If λ occurs r times, then you have r linearly independent vectors $u_1, \ldots, u_r$ that span the λ eigenspace. The Gram-Schmidt process allows you to construct an orthonormal set that spans this eigenspace, and while this orthonormal set isn't unique, any one of them will do.

For property 5, let e_i be the column vector that is zero except for a 1 in position i. The product $e_j^\top De_i = d_{ij}$. Let's write e_i and e_j in terms of the orthonormal basis $u_1, \ldots u_N$:

$$e_i = \sum_{k=1}^{N} (e_i \cdot u_k) u_k \text{ and } e_j = \sum_{k=1}^{N} (e_j \cdot u_k) u_k.$$

Using this expansion, we compute $e_j^\top De_i$ in a more complicated way:

$$e_j^\top De_i = \sum_{r=1}^{N}\sum_{s=1}^{N}(e_j \cdot u_r)(e_i \cdot u_s)(u_r^\top Du_s).$$

But $u_r^\top Du_s = \lambda_s(u_r \cdot u_s) = 0$ unless $r = s$, in which case it equals λ_r, so

$$e_j^\top De_i = \sum_{r=1}^{N}\lambda_r(e_j \cdot u_r)(e_i \cdot u_r).$$

On the other hand,

$$P^\top e_i = \begin{bmatrix}(e_i \cdot u_1)\\(e_i \cdot u_2)\\ \vdots \\(e_i \cdot u_N)\end{bmatrix}$$

and

$$\Lambda P^\top e_i = \begin{bmatrix}\lambda_1(e_i \cdot u_i)\\ \lambda_2(e_i \cdot u_2)\\ \vdots \\ \lambda_N(e_i \cdot u_N)\end{bmatrix}$$

Therefore the i, j entry of $P\Lambda P^\top$ is

$$(e_j^\top P)\Lambda(P^\top e_j) = \sum_{r=1}^{N}\lambda_r(e_i \cdot u_r)(e_j \cdot u_r) = d_{ij}$$

so the two matrices D and $P\Lambda P^\top$ are in fact equal.

Exercises:

1. Prove the rest of the first lemma in Section 2.4.2.
2. Prove the Gram-Schmidt Process has the claimed properties in Section 2.5.1.

3 Probability and Bayes Theorem

3.1 Introduction

Probability theory is one of the three central mathematical tools in machine learning, along with multivariable calculus and linear algebra. Tools from probability allow us to manage the uncertainty inherent in data collected from real world experiments, and to measure the reliability of predictions that we might make from that data. In these notes, we will review some of the basic terminology of probability and introduce Bayesian inference as a technique in machine learning problems.

This will only be a superficial introduction to ideas from probability. For a thorough treatment, see this open-source introduction to probability. For a more applied emphasis, I recommend the excellent online course Probabilistic Systems Analysis and Applied Probability and its associated text [2].

3.2 Probability Basics

The theory of probability begins with a set X of possible events or outcomes, together with a "probability" function P on (certain) subsets of X that measures "how likely" that combination of events is to occur.

The set X can be discrete or continuous. For example, when flipping a coin, our set of possible events would be the discrete set $\{H, T\}$ corresponding to the possible events of flipping heads or tails. When measuring the temperature using a thermometer, our set of possible outcomes might be the set of real numbers, or perhaps an interval in $\mathbb{R}$. The thermometer's measurement is random because it is affected by, say, electronic noise, and so its reading is the true temperature perturbed by a random amount.

The values of P are between 0, meaning that the event *will not* happen, and 1, meaning that it is certain to occur. As part of our set up, we assume that the total chance of some event from X occurring is 1, so that $P(X) = 1$; and the chance of "nothing" happening is zero, so $P(\emptyset) = 0$. And if $U \subset X$ is some collection, then $P(U)$ is the chance of an event from U occurring.

The last ingredient of this picture of probability is additivity. Namely, we assume that if U and V are subsets of X that are disjoint, then

$$P(U \cup V) = P(U) + P(V).$$

Even more generally, we assume that this holds for (countably) infinite collections of disjoint subsets $U_1, U_2, \ldots$, where

$$P(U_1 \cup U_2 \cup \cdots) = \sum_{i=1}^{\infty} P(U_i)$$

Definition: The combination of a set X of possible outcomes and a probability function P on subsets of X that satisfies $P(X) = 1$, $0 \le P(U) \le 1$ for all U, and is additive on countable disjoint collections of subsets of X is called a (naive) probability space. X is called the *sample space* and the subsets of X are called *events*.

Warning: The reason for the term "naive" in the above definition is that, if X is an uncountable set such as the real numbers $\mathbb{R}$, then the conditions in the definition are self-contradictory. This is a deep and rather surprising fact. To make a sensible definition of a probability space, one has to restrict the domain of the probability function P to certain subsets of X. These ideas form the basis of the mathematical subject known as measure theory. In these notes we will work with explicit probability functions and simple subsets such as intervals that avoid these technicalities.

3.2.1 Discrete probability examples

The simplest probability space arises in the analysis of coin-flipping. As mentioned earlier, the set X contains two elements $\{H, T\}$. The probability function P is determined by its value $P(\{H\}) = p$, where $0 \le p \le 1$, which is the chance of the coin yielding a "head". Since $P(X) = 1$, we have $P(\{T\}) = 1 - p$.

Other examples of discrete probability spaces arise from dice-rolling and playing cards. For example, suppose we roll two six-sided dice. There are 36 possible outcomes from this experiment, each equally likely. If instead we consider the sum of the two values on the dice, our outcomes range from 2 to 12 and the probabilities of these outcomes are given by

2	3	4	5	6	7	8	9	10	11	12
1/36	1/18	1/12	1/9	5/36	1/6	5/36	1/9	1/12	1/18	1/36

A traditional deck of 52 playing cards contains 4 aces. Assuming that the chance of drawing any card is the same (and is therefore equal to 1/52), the probability of drawing an ace is $4/52 = 1/13$ since

$$P(\{A_\clubsuit, A_\spadesuit, A_\heartsuit, A_\diamondsuit\}) = 4P(\{A_\clubsuit\}) = 4/52 = 1/13$$

3.2.2 Continuous probability examples

When the set X is continuous, such as in the temperature measurement, we measure $P(U)$, where $U \subset X$, by giving a "probability density function" $f : X \to \mathbb{R}$ and declaring that

$$P(U) = \int_U f(x)dX.$$

Notice that our function $f(x)$ has to satisfy the condition

$$P(X) = \int_X f(x)dX = 1.$$

For example, in our temperature measurement example, suppose the "true" outside temperature is t_0, and our thermometer gives a reading t. Then a good model for the random error is to assume that the error $x = t - t_0$ is governed by the density function

$$f_\sigma(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-x^2/2\sigma^2}$$

where σ is a parameter. In a continuous situation such as this one, the probability of any particular outcome in X is zero since

$$P(\{t\}) = \int_t^t f_\sigma(x)dx = 0$$

Still, the shape of the density function does tell you where the values are concentrated – values where the density function is larger are more likely than those where it is smaller.

With this density function, and x=$t - t_0$, the error in our measurement is given by

$$P(|t - t_0| < \delta) = \int_{-\delta}^{\delta} \frac{1}{\sigma\sqrt{2\pi}} e^{-x^2/2\sigma^2} dx \tag{3.1}$$

The parameter σ (called the *standard deviation*) controls how tightly the thermometer's measurement is clustered around the true value t_0; when σ is large, the measurements are scattered widely, when small, they are clustered tightly. See Figure 3.1.

3.3 Conditional Probability and Bayes Theorem

The theory of conditional probability gives a way to study how partial information about an event informs us about the event as a whole. For example, suppose you draw a card at random from a deck. As we've seen earlier, the chance that card is an ace is 1/13. Now suppose that you learn that (somehow) that the card is definitely not a jack, king, or queen. Since there

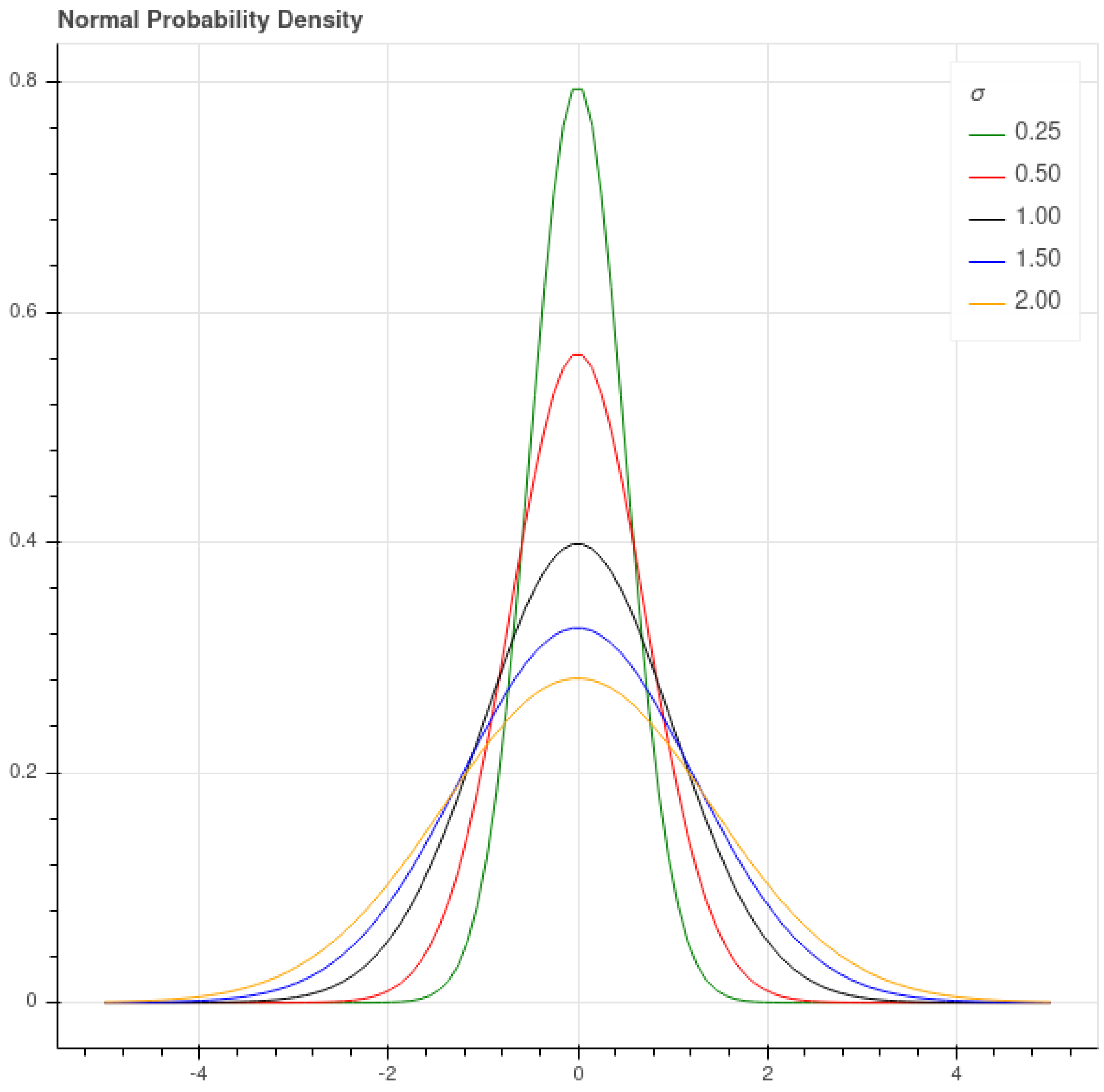


Figure 3.1: Normal Density

are 12 cards in the deck that are jacks, kings, or queens, the card you've drawn is one of the remaining 40 cards, which includes 4 aces. Thus the chance you are holding an ace is now $4/40 = 1/10$.

In terms of notation, if A is the event "my card is an ace" and B is the event "my card is not a jack, queen, or king" then we say that *the probability of A given B* is $1/10$. The notation for this is

$$P(A|B) = 1/10.$$

More generally, if A and B are events from a sample space X, and $P(B) > 0$, then

$$P(A|B) = \frac{P(A \cap B)}{P(B)},$$

so that $P(A|B)$ measures the chance that A occurs among those situations in which B occurs.

3.3.1 Bayes Theorem

Bayes theorem is a foundational result in probability.

Theorem: Bayes Theorem says

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}.$$

If we use the definition of conditional probability given above, this is straightforward:

$$\frac{P(B|A)P(A)}{P(B)} = \frac{P(B \cap A)}{P(B)} = P(A|B).$$

3.3.2 An example

To illustrate conditional probability, let's consider what happens when we administer the most reliable COVID-19 test, the PCR test, to an individual drawn from the population at large. There are two possible test results (positive and negative) and two possible true states of the person being tested (infected and not infected). Suppose I go to the doctor and get a COVID test which comes back positive. What is the probability that I actually have COVID?

Let's let S and W stand for infected (sick) and not infected (well), and let $+/-$ stand for test positive or negative. Note that there are four possible outcomes of our experiment:

- test positive and infected (S+) – this is a *true positive.*
- test positive and not infected (W+) – this is a *false positive.*
- test negative and infected (S-) – this is a *false negative.*

- test negative and not infected (W-) – this is a *true negative.*

The CDC says that the chance of a false positive – that is, the percentage of samples from well people that incorrectly yields a positive result – is about one-half of one percent, or 5 in 1000.

In other words,

$$P(+|W) = P(W+)/P(W) = 5/1000 = 1/200$$

On the other hand, the CDC tells us that chance of a false negative is 1 in 4, so

$$P(-|S) = P(S-)/P(S) = .25.$$

Since $P(S-) + P(S+) = P(S)$. since every test is either positive or negative, we have

$$P(+|S) = .75.$$

Suppose furthermore that the overall incidence of COVID-19 in the population is p. In other words, $P(S) = p$ so $P(W) = 1 - p$. Then

$$P(S+) = P(S)P(+|S) = .75p$$

and

$$P(W+) = P(W)P(+|W) = .005(1 - p).$$

Putting these together we get $P(+) = .005 + .745p$

What I'm interested in is $P(S|+)$ – the chance that I'm sick, given that my test result was positive. By Bayes Theorem,

$$P(S|+) = \frac{P(+|S)P(S)}{P(+)} = .75p/(.005 + .745p) = \frac{750p}{5 + 745p}.$$

As Figure 3.2 shows, if the population incidence is low then a positive test is far from conclusive. Indeed, if the overall incidence of COVID is one percent, then a positive test result only implies a 60 percent chance that I am in fact infected.

Just to fill out the picture, we have

$$P(-) = P(S-) + P(W-) = (P(S) - P(S+)) + (P(W) - P(W+))$$

which yields

$$P(-) = 1 - .005 + .005p - .75p = .995 - .745p.$$

Using Bayes Theorem, we obtain

$$P(S|-) = \frac{P(-|S)P(S)}{P(-)} = .25p/(.995 - .745p) = \frac{250p}{995 - 745p}.$$

In this case, even though the false negative rate is pretty high (25 percent) overall, if the population incidence is one percent, then the probability that you're sick given a negative result is only about .25 percent. So negative results are very likely correct!

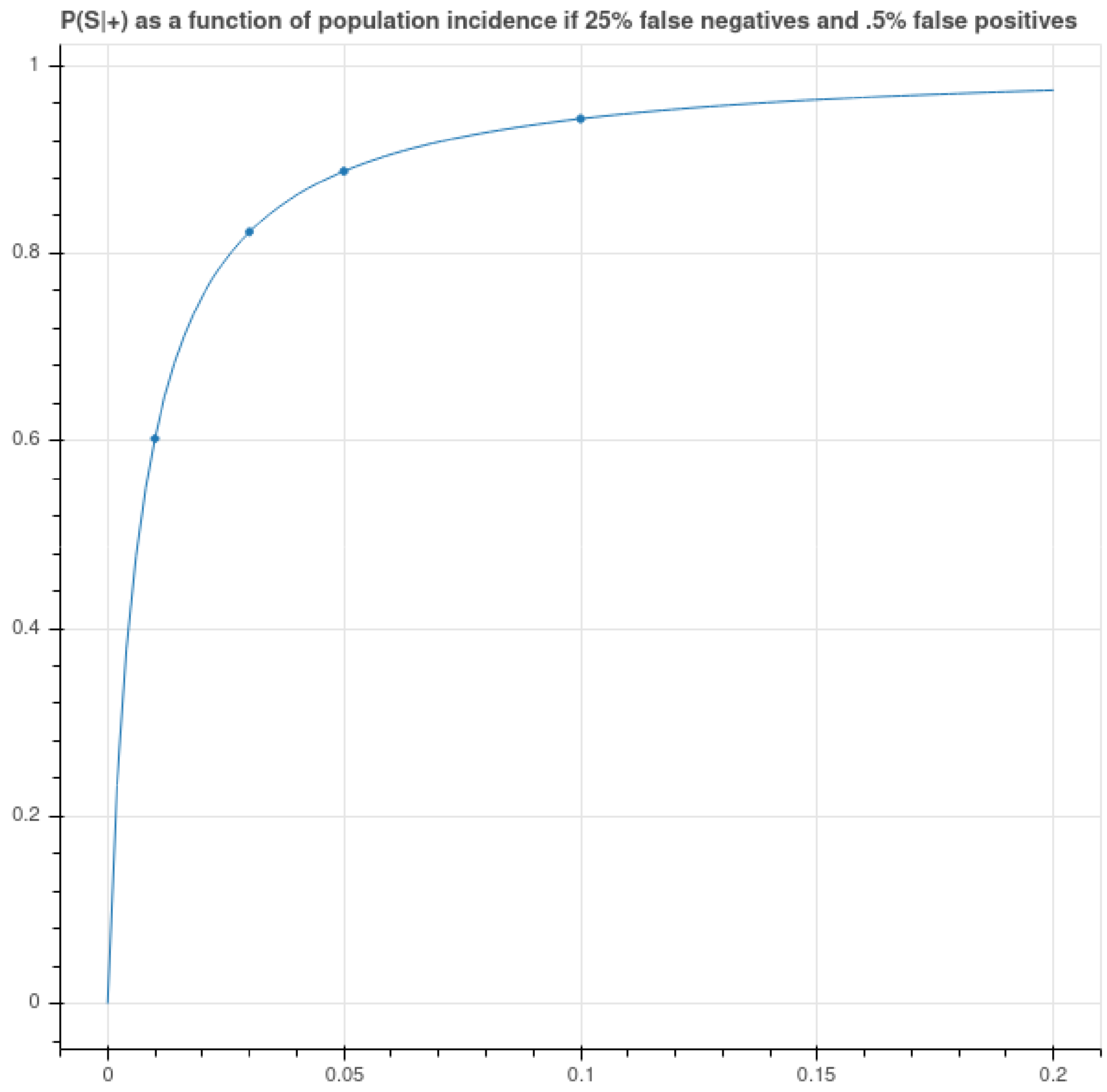


Figure 3.2: P(S|+) vs P(S)

3.4 Independence

Independence is one of the fundamental concepts in probability theory. Conceptually, two events are independent if the occurrence of one has does not influence the likelihood of the occurrence of the other. For example, successive flips of a coin are independent events, since the result of the second flip doesn't have anything to do with the result of the first. On the other hand, whether or not it rains today and tomorrow are not independent events, since the weather tomorrow depends (in a complicated way) on the weather today.

We can formalize this idea of independence using the following definition.

Definition: Let X be a sample space and let A and B be two events. Then A and B are *independent* if $P(A \cap B) = P(A)P(B)$. Equivalently, A and B are independent if $P(A|B) = P(A)$ and $P(B|A) = P(B)$.

3.4.1 Examples

3.4.1.1 Coin Flipping

Suppose our coin has a probability of heads given by a real number p between 0 and 1, and we flip our coin N times. What is the chance of gettting k heads, where $0 \leq k \leq N$? Any particular sequence of heads and tails containing k heads and $N-k$ tails has probability

$$P(\text{a particular sequence of } k \text{ heads among } N \text{ flips}) = p^k(1-p)^{N-k}.$$

In addition, there are $\binom{N}{k}$ sequences of heads and tails containing k heads. Thus the probability $P(k, N)$ of k heads among N flips is

$$P(k, N) = \binom{N}{k} p^k (1-p)^{N-k}. \tag{3.2}$$

Notice that the binomial theorem gives us $\sum_{k=0}^{N} P(k, N) = 1$ which is a reassuring check on our work.

The probability distribution on the set $X = \{0, 1, \ldots, N\}$ given by $P(k, N)$ is called the *binomial distribution* with parameters N and p.

3.4.1.2 A simple 'mixture'

Now let's look at an example of events that are not independent. Suppose that we have two coins, with probabilities of heads p_1 and p_2 respectively; and assume these probabilities are different. We play the a game in which we first choose one of the two coins (with equal chance) and then flip it twice. Is the result of the second flip independent of the first? In other words, is $P(HH) = P(H)^2$?

This type of situation is called a 'mixture distribution' because the probability of a head is a "mixture" of the probability coming from the two different coins.

The chance that the first flip is a head is $(p_1 + p_2)/2$ because it's the chance of picking the first coin, and then getting a head, plus the chance of picking the second, and then getting a head. The chance of getting two heads in a row is $(p_1^2 + p_2^2)/2$ because it's the chance, having picked the first coin, of getting two heads, plus the chance, having picked the second, of getting two heads.

Since

$$\frac{p_1^2 + p_2^2}{2} \neq \left(\frac{p_1 + p_2}{2}\right)^2$$

we see these events are not independent.

In terms of conditional probabilities, the chance that the second flip is a head, given that the first flip is, is computed as:

$$P(HH|H) = \frac{p_1^2 + p_2^2}{p_1 + p_2}.$$

From the Cauchy-Schwartz inequality one can show that

$$\frac{p_1^2 + p_2^2}{p_1 + p_2} > \frac{p_1 + p_2}{2}.$$

Why should this be? Why should the chance of getting a head on the second flip go up given that the first flip was a head? One way to think of this is that the first coin flip contains a little bit of information about which coin we chose. If, for example $p_1 > p_2$, and our first flip is heads, then it's just a bit more likely that we chose the first coin. As a result, the chance of getting another head is just a bit more likely than if we didn't have that information. We can make this precise by considering the conditional probability $P(p = p_1|H)$ that we've chosen the first coin given that we flipped a head. From Bayes' theorem:

$$P(p = p_1|H) = \frac{P(H|p = p_1)P(p = p_1)}{P(H)} = \frac{p_1}{p_1 + p_2} = \frac{1}{1 + (p_2/p_1)} > \frac{1}{2}$$

since $(1 + (p_2/p_1)) < 2$.

Exercise: Push this argument a bit further. Let $p_1 = \max(p_1, p_2)$ Let P_N be the conditional probability of getting heads assuming that the first N flips were heads. Show that $P_N \to p_1$

as $N \to \infty$. All those heads piling up make it more and more likely that you're flipping the first coin and so the chance of getting heads approaches p_1.

3.4.1.3 An example with a continuous distribution

Suppose that we return to our example of a thermometer which measures the ambient temperature with an error that is distributed according to the normal distribution, as in Equation 3.1. Suppose that we make 10 independent measurements $t_1, \dots, t_{10}$ of the true temperature t_0. What can we say about the distribution of these measurements?

In this case, independence means that

$$P = P(|t_1 - t_0| < \delta, |t_2 - t_0| < \delta, \dots) = P(|t_1 - t_0| < \delta)P(|t_2 - t_0| < \delta) \cdots P(|t_{10} - t_0| < \delta)$$

and therefore

$$P = \left(\frac{1}{\sigma\sqrt{2\pi}}\right)^{10} \int_{-\delta}^{\delta} \cdots \int_{-\delta}^{\delta} e^{-(\sum_{i=1}^{10} x_i^2)/2\sigma^2} dx_1 \cdots dx_{10}$$

One way to look at this is that the vector $\mathbf{e}$ of errors $(|t_1 - t_0|, \dots, |t_{10} - t_0|)$ is distributed according to a *multivariate gaussian distribution*:

$$P(\mathbf{e} \in U) = \left(\frac{1}{\sigma\sqrt{2\pi}}\right)^{10} \int_U e^{-\|x\|^2/2\sigma^2} d\mathbf{x} \tag{3.3}$$

where U is a region in $\mathbf{R}^{10}$.

The multivariate gaussian can also describe situations where independence does not hold. For simplicity, let's work in two dimensions and consider the probability density on $\mathbf{R}^2$ given by

$$P(\mathbf{e} \in U) = A \int_U e^{-(x_1^2 - x_1 x_2 + x_2^2)/2\sigma^2} d\mathbf{x}.$$

where the constant A is chosen so that

$$A \int_{\mathbf{R}^2} e^{-(x_1^2 - x_1 x_2 + x_2^2)/2\sigma^2} d\mathbf{x} = 1.$$

This density function as a "bump" concentrated near the origin in $\mathbf{R}^2$, and its level curves are a family of ellipses centered at the origin. See Figure 3.3 for a plot of this function with $\sigma = 1$.

In this situation we can look at the conditional probability of the first variable given the second, and see that the two variables are not independent. Indeed, if we fix x_2, then the distribution of x_1 depends on our choice of x_2. We could see this by a calculation, or we can just look at the graph: if $x_2 = 0$, then the most likely values of x_1 cluster near zero, while if $x_2 = 1$, then the most likely values of x_1 cluster somewhere above zero.

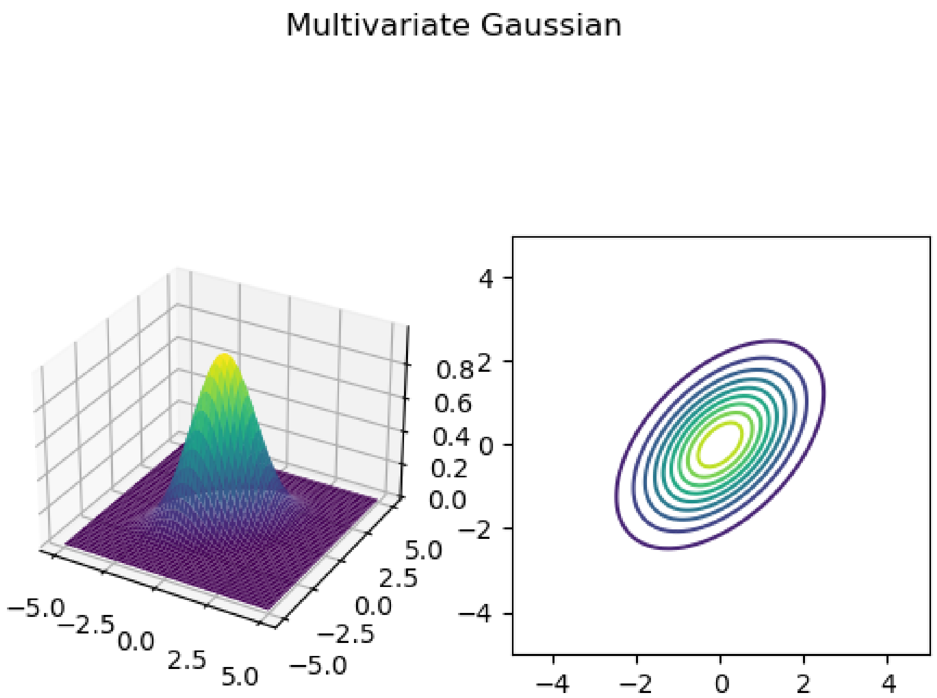


Figure 3.3: Multivariate Gaussian

3.5 Random Variables, Mean, and Variance

Typically, when we are studying a random process, we aren't necessarily accessing the underlying events, but rather we are making measurements that provide us with some information about the underlying events. For example, suppose our sample space X is the set of throws of a pair of dice, so X contains the 36 possible combinations that can arise from the throws. What we are actually interested is the sum of the values of the two dice – that's our "measurement" of this system. This rather vague notion of a measurement of a random system is captured by the very general idea of a *random variable*.

Definition: Let X be a sample space with probability function P. A *random variable* on X is a function $f : X \to \mathbb{R}$.

Given a random variable f, we can use the probability measure to decide how likely f is to take a particular value, or values in a particular set by the formula

$$P(f(x) \in U) = P(f^{-1}(U))$$

In the dice rolling example, the random variable S that assigns their sum to the pair of values obtained on two dice is a random variable. Those values lie between 2 and 12 and we have

$$P(S = k) = P(S^{-1}(\{k\})) = P(\{(x, y) : x + y = k\})$$

where (x, y) runs through $\{1, 2, \ldots, 6\}^2$ representing the two values and $P((x, y)) = 1/36$ since all throws are equally likely.

Let's look at a few more examples, starting with what is probably the most fundamental of all.

Definition: Let X be a sample space with two elements, say H and T, and suppose that $P(H) = p$ for some $0 \leq p \leq 1$. Then the random variable that satisfies $f(H) = 1$ and $f(T) = 0$ is called a Bernoulli random variable with parameter p.

In other words, a Bernoulli random variable gives the value 1 when a coin flip is heads, and 0 for tails.

Now let's look at what we earlier called the binomial distribution.

Definition: Let X be a sample space consisting of strings of H and T of length N, with the probability of a *particular string* S with k heads and $N - k$ tails given by

$$P(S) = p^k(1 - p)^{N-k}$$

for some $0 \leq p \leq 1$. In other words, X is the sample space consisting of N independent flips of a coin with probability of heads given by p.

Let $f : X \to \mathbb{R}$ be the function which counts the number of H in the string. We can express f in terms of Bernoulli random variables; indeed,

$$f = X_1 + ... + X_N$$

where each X_i is a Bernoulli random variable with parameter p.

Now

$$P(f = k) = \binom{N}{k} p^k (1-p)^{N-k}$$

since $f^{-1}(\{k\})$ is the number of elements in the subset of strings of H and T of length N containing exactly k H's. This is our old friend the binomial distribution. So *a binomial distribution is the distribution of the sum of N independent Bernoulli random variables.*

For an example with a continuous random variable, suppose our sample space is $\mathbf{R}^2$ and the probability density is the simple multivariate normal

$$P(\mathbf{x} \in U) = \left(\frac{1}{\sqrt{2\pi}}\right)^2 \int_U e^{-\|\mathbf{x}\|^2/2} d\mathbf{x}.$$

Let f be the random variable $f(\mathbf{x}) = \|\mathbf{x}\|$. The function f measures the Euclidean distance of a randomly drawn point from the origin. The set

$$U = f^{-1}([0, r)) \subseteq \mathbf{R}^2$$

is the circle of radius r in $\mathbf{R}^2$. The probability that a randomly drawn point lies in this circle is

$$P(f < r) = \left(\frac{1}{\sqrt{2\pi}}\right)^2 \int_U e^{-\|\mathbf{x}\|^2/2} d\mathbf{x}.$$

We can actually evaluate this integral in closed form by using polar coordinates. We obtain

$$P(f < r) = \left(\frac{1}{\sqrt{2\pi}}\right)^2 \int_{\theta=0}^{2\pi} \int_{\rho=0}^{r} e^{-\rho^2/2} \rho \, d\rho \, d\theta.$$

Since

$$\frac{d}{d\rho} e^{-\rho^2/2} = -\rho e^{-\rho^2/2}$$

we have

$$\begin{aligned} P(f < r) &= -\frac{1}{2\pi} \theta e^{-\rho^2/2} |_{\theta=0}^{2\pi} |_{\rho=0}^{r} \\ &= 1 - e^{-r^2/2} \end{aligned}$$

The probability density associated with this random variable is the derivative of $1 - e^{-r^2/2}$

$$P(f \in [a,b]) = \int_{r=a}^{b} re^{-r^2/2}dr$$

as you can see by the fundamental theorem of calculus. This density is drawn in Figure 3.4 where you can see that the points are clustered at a distance of 1 from the origin.

3.5.1 Independence and Random Variables

We can extend the notion of independence from events to random variables.

Definition: Let f and g be two random variables on a sample space X with probability P. Then f and g are independent if, for all intervals U and V in $\mathbb{R}$, the events $f^{-1}(U)$ and $g^{-1}(V)$ are independent.

For discrete probability distributions, this means that, for all $a, b \in \mathbb{R}$,

$$P(f = a \text{ and } g = b) = P(f = a)P(g = b).$$

For continous probability distributions given by a density function $P(x)$, independence can be more complicated to figure out.

3.5.2 Expectation, Mean and Variance

The most fundamental tool in the study of random variables is the concept of "expectation", which is a fancy version of average. The word "mean" is a synonym for expectation – the mean of a random variable is the same as its expectation or "expected value."

Definition: Let X be a sample space with probability measure P. Let $f : X \to \mathbb{R}$ be a random variable. Then the *expectation* or *expected value* $E[f]$ of f is

$$E[f] = \int_X f(x)dP.$$

More specifically, if X is discrete, then

$$E[f] = \sum_{x \in X} f(x)P(x)$$

while if X is continuous with probability density function $p(x)dx$ then

$$E[f] = \int_X f(x)p(x)dx.$$

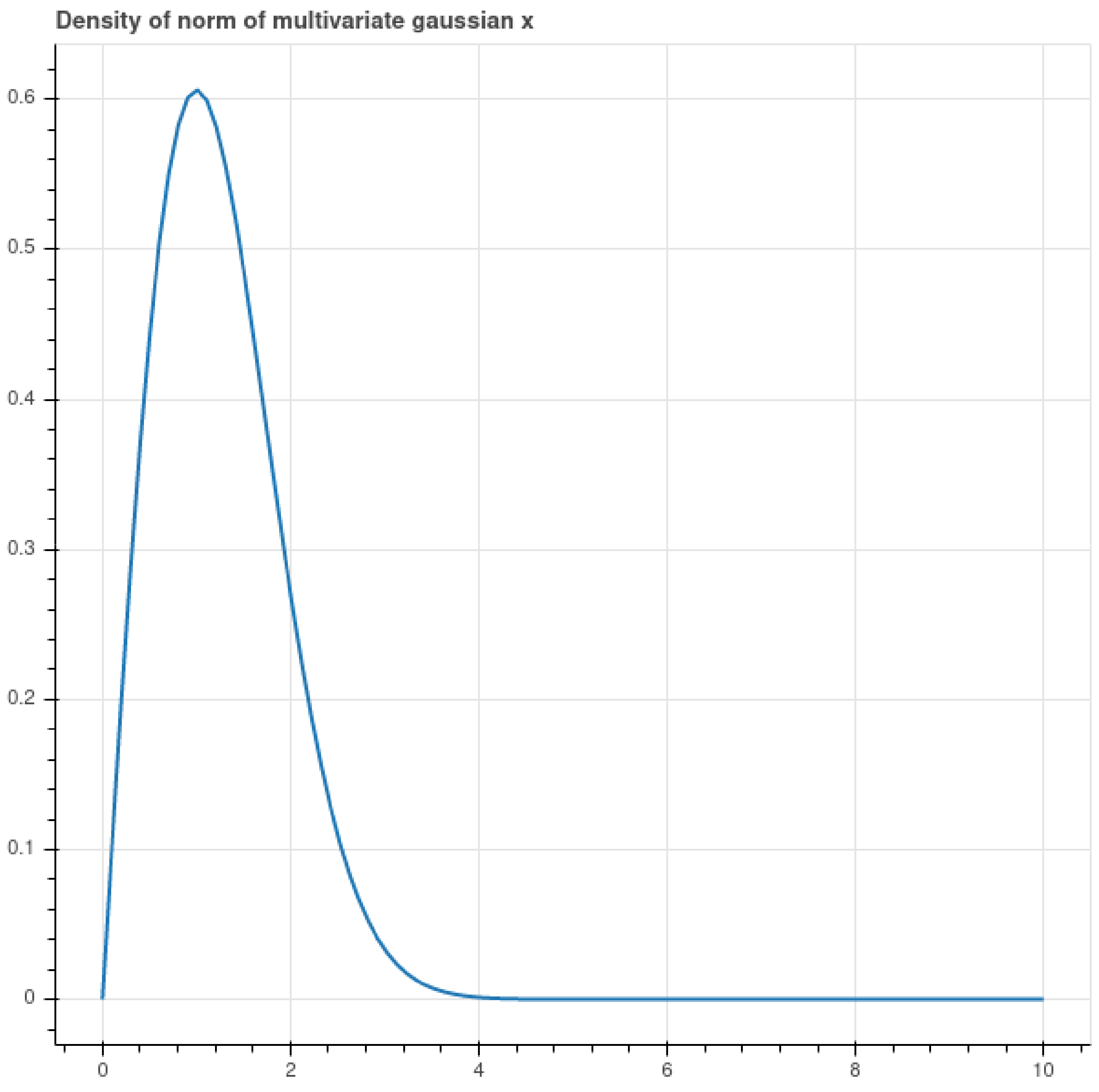


Figure 3.4: Density of the Norm

If f is a Bernoulli random variable with parameter p, then

$$E[f] = 1 \cdot p + 0 \cdot (1-p) = p$$

If f is a binomial random variable with parameters p and N, then

$$E[f] = \sum_{i=0}^{N} i \binom{N}{i} p^i (1-p)^{N-i}$$

One can evaluate this using some combinatorial tricks, but it's easier to apply this basic fact about expectations.

Proposition: Expectation is linear: $E[aX + bY] = aE[X] + bE[Y]$ for random variables X, Y and constants a and b.

The proof is an easy consequence of the expression of E as a sum (or integral).

Since a binomial random variable Z with parameters N and p is the sum of N Bernoulli random variables, its expectation is

$$E[X_1 + \cdots + X_N] = Np.$$

A more sophisticated property of expectation is that it is multiplicative when the random variables are independent.

Proposition: Let f and g be two independent random variables. Then $E[fg] = E[f]E[g]$.

Proof: Let's suppose that the sample space X is discrete. By definition,

$$E[f] = \sum_{x \in X} f(x)P(x)$$

and we can rewrite this as

$$E[f] = \sum_{a \in \mathbf{R}} aP(\{x : f(x) = a\}).$$

Let $Z \subset \mathbb{R}$ be the range of f. Then

$$\begin{aligned}
E[fg] &= \sum_{a \in Z} aP(\{x : fg(x) = a\}) \\
&= \sum_{a \in Z} \sum_{\substack{(u,v) \in \mathbf{Z}^2 \\ uv=a}} aP(\{x : f(x) = u \text{ and } g(x) = v\}) \\
&= \sum_{a \in Z} \sum_{\substack{\mathbf{Z}^2 \\ uv=a}} uvP(\{x : f(x) = u\})P(\{x : g(x) = v\}) \\
&= \sum_{u \in Z} uP(\{x : f(x) = u\}) \sum_{v \in Z} vP(\{x : f(x) = v\}) \\
&= E[f]E[g]
\end{aligned}$$

3.5.2.1 Variance

The variance of a random variable is a measure of its dispersion around its mean.

Definition: Let f be a random variable. Then the variance is the expression

$$\sigma^2(f) = E[(f - E[f])^2] = E[f^2] - (E[f])^2$$

The square root of the variance is called the "standard deviation."

The two formulae for the variance arise from the calculation

$$E[(f - E[f])^2] = E[(f^2 - 2fE[f] + E[f]^2)] = E[f^2] - 2E[f]^2 + E[f]^2 = E[f^2] - E[f]^2.$$

To compute the variance of the Bernoulli random variable f with parameter p, we first compute

$$E[f^2] = p(1)^2 + (1-p)0^2 = p.$$

Since $E[f] = p$, we have

$$\sigma^2(f) = p - p^2 = p(1-p).$$

If f is the binomial random variable with parameters N and p, we can again use the fact that f is the sum of N Bernoulli random variables $X_1 + \cdots + X_n$ and compute

$$\begin{aligned} E[(\sum_i X_i)^2] - E[\sum_i X_i]^2 &= E[\sum_i X_i^2 + \sum_{i,j} X_i X_j] - N^2p^2 \\ &= Np + N(N-1)p^2 - N^2p^2 \\ &= Np(1-p) \end{aligned}$$

where we have used the fact that the square X^2 of a Bernoulli random variable is equal to X.

For a continuous example, suppose that we consider a sample space $\mathbb{R}$ with the normal probability density

$$P(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-x^2/2\sigma^2} dx.$$

The mean of the random variable x is

$$E[x] = \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^{\infty} x e^{-x^2/2\sigma^2} dx = 0$$

since the function being integrated is odd. The variance is

$$E[x^2] = \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^{\infty} x^2 e^{-x^2/2\sigma^2} dx.$$

The trick to evaluating this integral is to consider the derivative:

$$\frac{d}{d\sigma}\left[\frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-x^2/(2\sigma^2)} dx\right] = 0$$

where the result is zero since the quantity being differentiated is a constant (namely 1). Sorting through the resulting equation leads to the fact that

$$E[x^2] = \sigma^2$$

so that the σ^2 parameter in the normal distribution really *is* the variance of the associated random variable.

3.6 Models and Likelihood

A *statistical model* is a mathematical model that accounts for data via a process that incorporates random behavior in a structured way. We have seen several examples of such models in our discussion so far. For example, the Bernoulli process that describes the outcome of a series of coin flips as independent choices of heads or tails with probability p is a simple statistical model; our more complicated mixture model in which we choose one of two coins at random and then flip that is a more complicated model.
Our description of the variation in temperature measurements as arising from perturbations from the true temperature by a normally distributed amount is another example of a statistical model, this one involving a continuous random variable.

When we apply a mathematical model to understand data, we often have a variety of parameters in the model that we must adjust to get the model to best "fit" the observed data. For example, suppose that we observe the vibrations of a block attached to a spring. We know that the motion is governed by a second order linear differential equation, but the dynamics depend on the mass of the block, the spring constant, and the damping coefficient. By measuring the dynamics of the block over time, we can try to work backwards to figure out these parameters, after which we will be able to predict the block's motion into the future.

3.6.1 Maximum Likelihood (Discrete Case)

To see this process in a statistical setting, let's return to the simple example of a coin flip. The only parameter in our model is the probability p of getting heads on a particular flip. Suppose that we flip the coin 100 times and get 55 heads and 45 tails. What can we say about p?

We will approach this question via the "likelihood" function for our data. We ask: for a particular value of the parameter p, how likely is this outcome? From Equation 3.2 we have

$$P(55H, 45T) = \binom{100}{55} p^{55}(1-p)^{45}.$$

This function is plotted in Figure 3.5. As you can see from that plot, it is extremely unlikely that we would have gotten 55 heads if p was smaller than .4 or greater than .7, while the *most likely* value of p occurs at the maximum value of this function, and a little calculus tells us that this point is where $p = .55$. This *most likely* value of p is called the *maximum likelihood estimate* for the parameter p.

3.6.2 Maximum Likelihood (Continuous Case)

Now let's look at our temperature measurements where the error is normally distributed with variance parameter σ^2. As we have seen earlier, the probability density of errors $\mathbf{x} = (x_1, \ldots, x_n)$ of n independent measurements is

$$P(\mathbf{x}) = \left(\frac{1}{\sigma\sqrt{2\pi}}\right)^n e^{-\|\mathbf{x}\|^2/(2\sigma^2)} d\mathbf{x}.$$

(see Equation 3.3). What should we use as the parameter σ? We can ask which choice of σ makes our data *most likely.* To calculate this, we think of the probability of a function of σ and use Calculus to find the maximum. It's easier to do this with the logarithm.

$$\log P(\mathbf{x}) = \frac{-\|\mathbf{x}\|^2}{2\sigma^2} - n\log\sigma + C$$

where C is a constant that we'll ignore. Taking the derivative and setting it to zero, we obtain

$$-\|\mathbf{x}\|^2\sigma^{-3} - n\sigma^{-1} = 0$$

which gives the formula

$$\sigma^2 = \frac{\|\mathbf{x}\|^2}{n}$$

This should look familiar! The maximum likelihood estimate of the variance is the *mean-squared-error.*

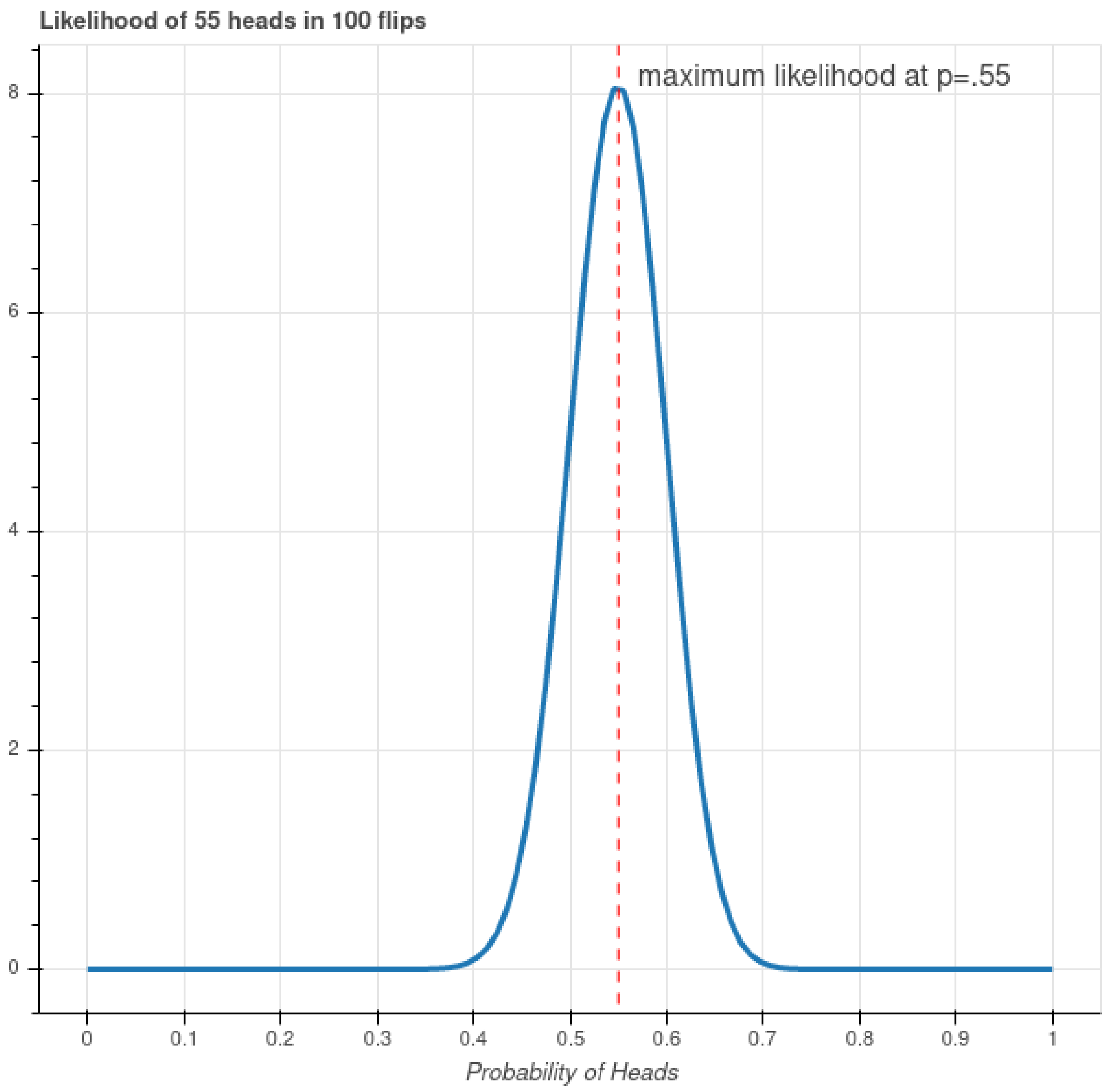


Figure 3.5: Likelihood Plot

3.6.3 Linear Regression and likelihood

In our earlier lectures we discussed linear regression at length. Our introduction of ideas from probability give us new insight into this fundamental tool. Consider a statistical model in which certain measured values y depend linearly on x up to a normally distributed error:

$$y = mx + b + \epsilon$$

where ϵ is drawn from the normal distribution with variance σ^2.

The classic regression setting has us measuring a collection of N points (x_i, y_i) and then asking for the "best" m, b, and σ^2 to explain these measurements. Using the likelihood perspective, each value $y_i - mx_i - b$ is an independent draw from the normal distribution with variance σ^2, exactly like our temperature measurements in the one variable case.

The likelihood (density) of those draws is therefore

$$P = \left(\frac{1}{\sigma\sqrt{2\pi}}\right)^N e^{-\sum_i (y_i - mx_i - b)^2/(2\sigma^2)}.$$

What is the maximum likelihood estimate of the parameters m, b, and σ^2?

To find this we look at the logarithm of P and take derivatives.

$$\log(P) = -N\log(\sigma) - \frac{1}{2\sigma^2}\sum_i (y_i - mx_i - b)^2.$$

As far as m and b are concerned, the minimum comes from the derivatives with respect to m and b of

$$\sum_i (y_i - mx_i - b)^2.$$

In other words, the maximum likelihood estimate m_* and b_* for m and b are *exactly the ordinary least squares estimates.*

As far as σ^2 is concerned, we find just as above that the maximum likelihood estimate σ_*^2 is the mean squared error

$$\sigma_*^2 = \frac{1}{N}\sum_i (y_i - m_* x_i - b_*)^2.$$

The multivariate case of regression proposes a model of the form

$$Y = X\beta + \epsilon$$

and a similar calculation again shows that the least squares estimates for β are the maximum likelihood values for this model.

3.7 Bayesian Inference

We conclude our review of ideas from probability by examining the Bayesian perspective on data.

Suppose that we wish to conduct an experiment to determine the temperature outside our house. We begin our experiment with a statistical model that is supposed to explain the variability in the results. The model depends on some parameters that we wish to estimate. For example, the parameters of our experiment might be the 'true' temperature t_* and the variance σ^2 of the error.

From the Bayesian point of view, at the beginning of this experiment we have an initial sense of what the temperature is likely to be, expressed in the form of a probability distribution. This initial information is called the *prior* distribution.

For example, if we know that it's December in Connecticut, our prior distribution might say that the temperature is more likely to be between 20 and 40 degrees Fahrenheit and is quite unlikely to be higher than 60 or lower than 0. So our prior distribution might look like Figure 3.6.

If we really have no opinion about the temperature other than its between say, -20 and 100 degrees, our prior distribution might be uniform over that range, as in Figure 3.7.

The choice of a prior will guide the interpretation of our experiments in ways that we will see shortly.

The next step in our experiment is the collection of data. Suppose we let $\mathbf{t} = (t_1, t_2, \dots, t_n)$ be a random variable representing n independent measurements of the temperature. We consider the *joint distribution* of the parameters t_* and σ^2 and the possible measurements $\mathbf{t}$:

$$P(\mathbf{t}, t_*, \sigma^2) = \left(\frac{1}{\sigma\sqrt{2\pi}}\right)^n e^{-\|\mathbf{t}-t_*\mathbf{e}\|^2/(2\sigma^2)}$$

where $\mathbf{e} = (1, 1, \dots, 1)$.

The conditional probability $P(t_*, \sigma^2|\mathbf{t})$ is the distribution of the values of t_* and $\sigma^2 *given*$a value of the $\mathbf{t}$. This is what we hope to learn by our experiment – namely, if we make a particular measurement, what does it tell us about t_* and σ^2?

Now suppose that we actually make some measurements, and so we obtain a specific set of values $\mathbf{t}_0$ for $\mathbf{t}$.

By Bayes Theorem,

$$P(t_*, \sigma^2|\mathbf{t} = \mathbf{t}_0) = \frac{P(\mathbf{t} = \mathbf{t}_0|t_*, \sigma^2)P(t_*, \sigma^2)}{P(\mathbf{t} = \mathbf{t}_0)}$$

We interpret this as follows:

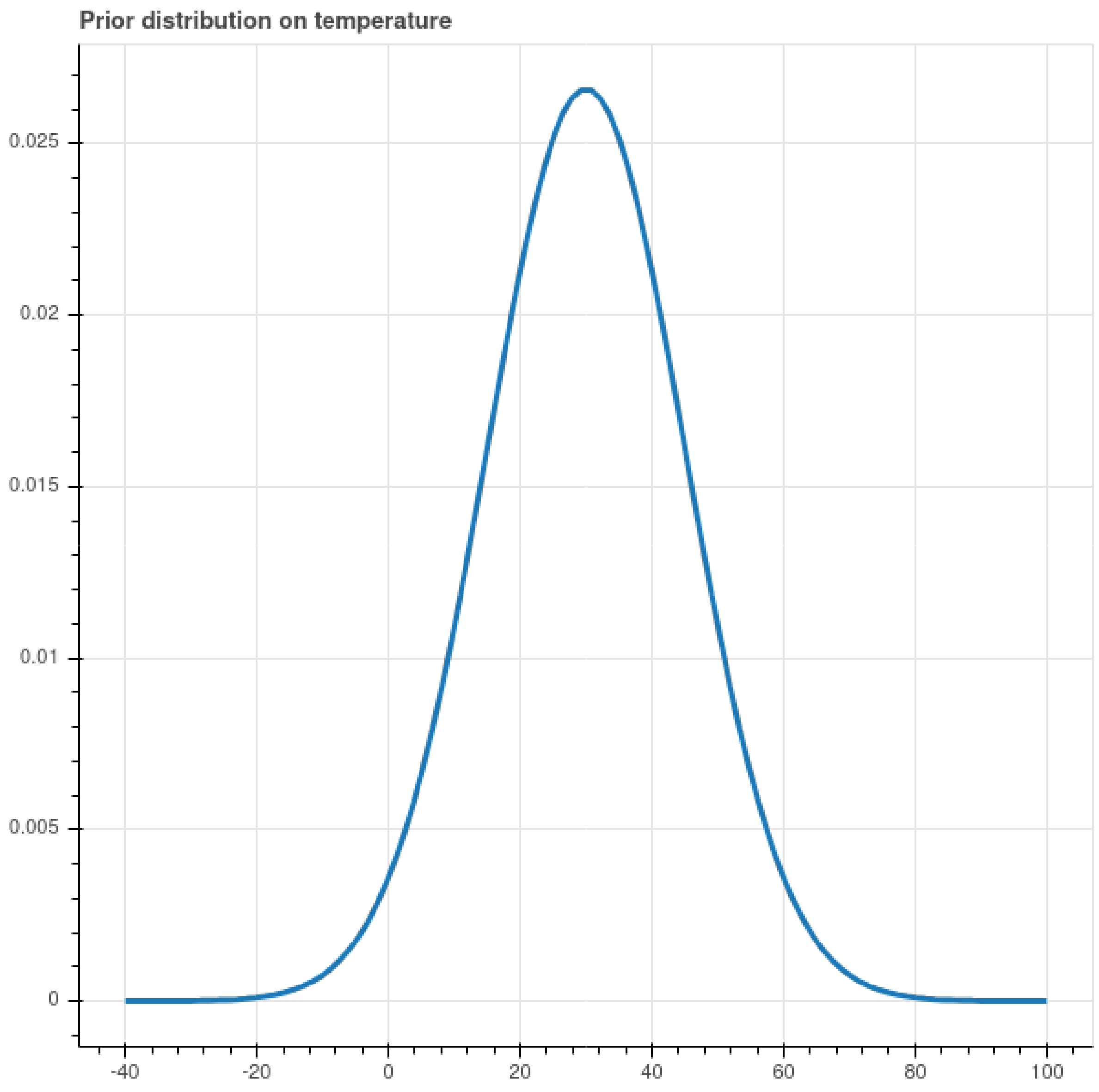


Figure 3.6: Prior Distribution on Temperature

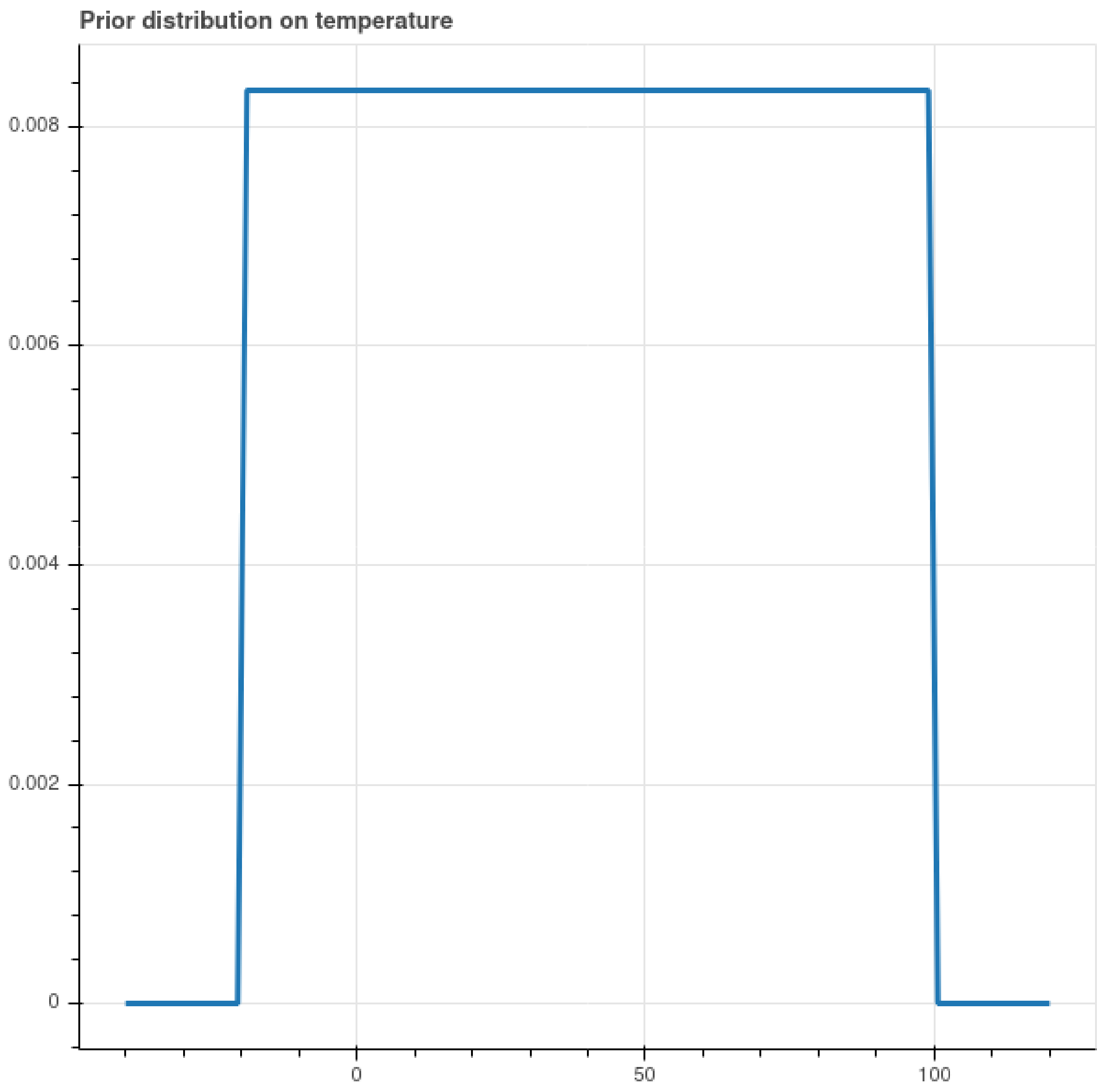


Figure 3.7: Uniform Prior

- the left hand side $P(t_*, \sigma^2 | \mathbf{t} = \mathbf{t}_0)$ is called the *posterior distribution* and is the distribution of t_* and σ^2 obtained by *updating our prior knowledge with the results of our experiment.*
- The probability $P(\mathbf{t} = \mathbf{t}_0 | t_*, \sigma^2)$ is the probability of obtaining the measurements we found for a particular value of the parameters t_* and σ^2.
- The probability $P(t_*, \sigma^2)$ is the *prior distribution* on the parameters that reflects our initial impression of the distribution of these parameters.
- The denominator $P(\mathbf{t} = \mathbf{t}_0)$ is the total probability of the results that we obtained, and is the integral over the distribution of the parameters weighted by their prior probability:

$$P(\mathbf{t} = \mathbf{t}_0) = \int_{t_*, \sigma^2} P(\mathbf{t} = \mathbf{t}_0 | t_*, \sigma^2) P(t_*, \sigma^2)$$

3.7.1 Bayesian experiments with the normal distribution

To illustrate these Bayesian ideas, we'll consider the problem of measuring the temperature, but for simplicity let's assume that we fix the variance in our error measurements at 1 degree. Let's use the prior distribution on the true temperature that I proposed in Figure 3.6, which is a normal distribution with variance 15 "shifted" to be centered at 30:

$$P(t_*) = \left(\frac{1}{\sqrt{2\pi}}\right) e^{-(t_* - 30)^2/30}.$$

The expected value $E[t]$ – the mean of the this distribution – is 30.

Since the error in our measurements is normally distributed with variance 1, we have

$$P(t - t_*) = \left(\frac{1}{\sqrt{2\pi}}\right) e^{-(t - t_*)^2/2}$$

or as a function of the absolute temperature, we have

$$P(t, t_*) = \left(\frac{1}{\sqrt{2\pi}}\right) e^{-(t - t_*)^2/2}.$$

Now we make a bunch of measurements to obtain $\mathbf{t}_0 = (t_1, \ldots, t_n)$. We have

$$P(\mathbf{t} = \mathbf{t}_0 | t_*) = \left(\frac{1}{\sqrt{2\pi}}\right)^n e^{-\|\mathbf{t} - t_* \mathbf{e}\|^2/2}.$$

The total probability $T = P(\mathbf{t} = \mathbf{t_0})$ is hard to calculate, so let's table that for a while. The posterior probability is

$$P(t_* | \mathbf{t} = \mathbf{t}_0) = \frac{1}{T} \left(\frac{1}{\sqrt{2\pi}}\right)^n e^{-\|\mathbf{t} - t_* \mathbf{e}\|^2/2} \left(\frac{1}{\sqrt{2\pi}}\right) e^{-(t_* - 30)^2/30}.$$

Leaving aside the multiplicative constants for the moment, consider the exponential

$$e^{-(\|\mathbf{t}-t_*\mathbf{e}\|^2/2+(t_*-30)^2)/30}.$$

Since $\mathbf{t}$ is a vector of constants – it is a vector of our particular measurements – the exponent

$$\|\mathbf{t}-t_*\mathbf{e}\|^2/2+(t_*-30)^2/30=(t_*-30)^2/30+\sum_i(t_i-t_*)^2/2$$

is a quadratic polynomial in t_* that simplifies:

$$(t_*-30)^2/30+\sum_i(t_i-t_*)^2/2=At_*^2+Bt_*+C.$$

Here

$$A=(\frac{1}{30}+\frac{n}{2}),$$
$$B=-2-\sum_i t_i$$
$$C=30+\frac{1}{2}\sum_i t_i^2.$$

We can complete the square to write

$$At_*^2+Bt_*+C=(t_*-U)^2/2V+K$$

where

$$U=\frac{2+\sum_i t_i}{\frac{1}{15}+n}$$

and

$$V=\frac{1}{\frac{1}{15}+n}.$$

So up to constants that don't involve t_*, the posterior density is of the form

$$e^{(t_*-U)^2/2V}$$

and since it is a probability density, the constants must work out to give total integral of 1. Therefore the posterior density is a normal distribution centered at U and with variance V. Here U is called the*posterior mean*and V the*posterior variance.*

To make this explicit, suppose $n=5$ and we measured the following temperatures:

$$40, 41, 39, 37, 44$$

The mean of these observations is 40.2 and the variance is 5.4.

A calculation shows that the posterior mean is 40.1 and the posterior variance is 0.2. Comparing the prior with the posterior, we obtain the plot in Figure 3.8. The posterior has a sharp peak at 40.1 degrees. This value is just a bit smaller than the mean of the observed temperatures which is 40.2 degrees. This difference is caused by the prior – our prior distribution said the temperature was likely to be around 30 degrees, and so the prior pulls the observed mean a bit towards the prior mean taking into account past experience. Because the variance of the prior is large, it has a relatively small influence on the posterior.

The general version of the calculation above is summarized in this proposition.

Proposition: Suppose that our statistical model for an experiment proposes that the measurements are normally distributed around an (unknown) mean value of μ with a (fixed) variance σ^2. Suppose further that our prior distribution on the unknown mean μ is normal with mean μ_0 and variance τ^2. Suppose we make measurements

$$y_1, \cdots, y_n$$

with mean $\overline{y}$. Then the posterior distribution of μ is again normal, with posterior variance

$$\tau'^2 = \frac{1}{\frac{1}{\tau^2} + \frac{n}{\sigma^2}}$$

and posterior mean

$$\mu' = \frac{\frac{\mu_0}{\tau^2} + \frac{n}{\sigma^2}\overline{y}}{\frac{1}{\frac{1}{\tau^2}+\frac{n}{\sigma^2}}}$$

So the posterior mean is a sort of weighted average of the sample mean and the prior mean; and as $n \to \infty$, the posterior mean approaches the sample mean – in other words, as you get more data, the prior has less and less influence on the results of the experiment.

3.7.2 Bayesian coin flipping

For our final example in this fast overview of ideas from probability, we consider the problem of deciding whether a coin is fair. Our experiment consists of N flips of a coin with unknown probability p of heads, so the data consists of the number h of heads out of the N flips. To apply Bayesian reasoning, we need a prior distribution on p. Let's first assume that we have no reason to prefer one value of p over another, and so we choose for our prior the uniform distribution on p between 0 and 1.

We wish to analyze $P(p|h)$, the probability distribution of p given h heads out of N flips. Bayes Theorem gives us:

$$P(p|h) = \frac{P(h|p)P(p)}{P(h)}$$

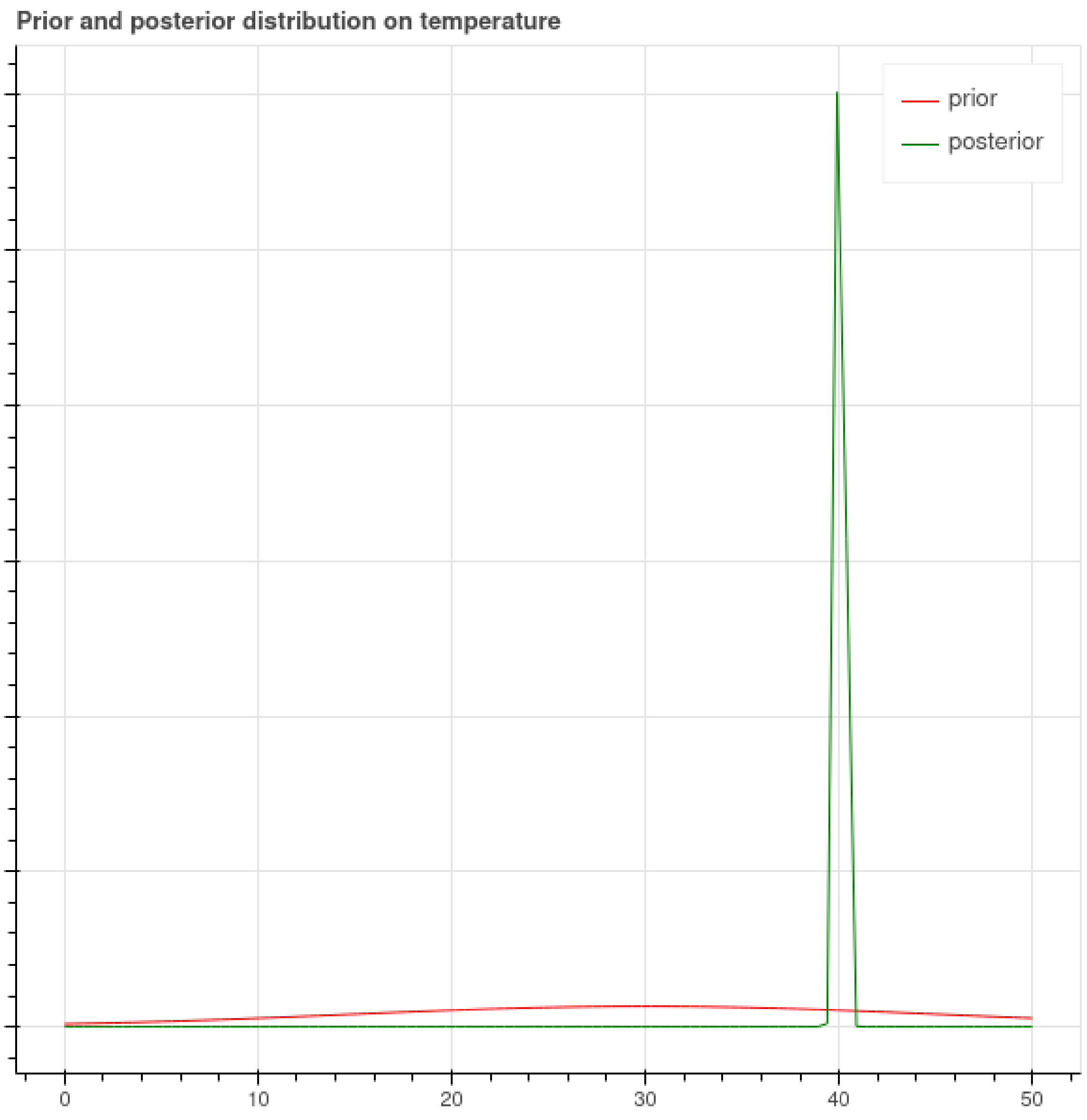


Figure 3.8: Prior and Posterior

where

$$P(h|p) = \binom{N}{h} p^h (1-p)^{N-h}$$

and

$$P(h) = \int_{p=0}^{1} P(h|p)P(p)dp = \binom{N}{h} \int_{p=0}^{1} p^h (1-p)^{N-h} dp$$

is a constant which insures that

$$\int_p P(p|h)dp = 1.$$

We see that the posterior distribution $P(p|h)$ is proportional to the polynomial function

$$P(p|h) \propto p^h (1-p)^{N-h}.$$

As in Section 3.6.1, we see that this function peaks at h/N. This is called the maximum *a posteriori estimate* for p.

Another way to summarize the posterior distribution $P(p|h)$ is to look at the expected value of p. This is called the *posterior mean* of p. To compute it, we need to know the normalization constant in the expression for $P(p|h)$, and for that we can take advantage of the properties of a special function $B(a,b)$ called the Beta-function:

$$B(a,b) = \int_{p=0}^{1} p^{a-1}(1-p)^{b-1} dp.$$

Proposition: If a and b are integers, then $B(a,b) = \frac{a+b}{ab}\frac{1}{\binom{a+b}{a}}$.

Proof: Using integration by parts, one can show that

$$B(a,b) = \frac{a-1}{b} B(a-1,b+1)$$

and a simple calculation shows that

$$B(1,b) = \frac{1}{b}.$$

Let

$$H(a,b) = \frac{a+b}{ab}\frac{1}{\binom{a+b}{a}} = \frac{(a-1)!(b-1)!}{(a+b-1)!}$$

Then it's easy to check that H satsifies the same recurrences as $B(a,b)$, and that $H(1,b) = 1/b$. So the two functions agree by induction.

Using this Proposition, we see that

$$P(p|h) = \frac{p^h (1-p)^{N-h}}{B(h+1, N-h+1)}$$

and

$$E[p] = \frac{\int_{p=0}^{1} p^{h+1}(1-p)^{N-h}dp}{B(h+1, N-h+1)} = \frac{B(h+2, N-h+1)}{B(h+1, N-h+1)}.$$

Sorting through this using the formula for $B(a, b)$ we obtain

$$E[p] = \frac{h+1}{N+2}.$$

So if we obtained 55 heads out of 100 flips, the maximum a posteriori estimate for p is .55, while the posterior mean is $56/102 = .549$ – just a bit less.

Now suppose that we had some reason to believe that our coin was fair. Then we can choose a prior probability distribution that expresses this. For example, we can choose

$$P(p) = \frac{1}{B(5,5)}p^4(1-p)^4.$$

Here we use the Beta function to guarantee that $\int_0^1 P(p)dp = 1$. We show this prior distribution in Figure 3.9.

If we redo our Bayes theorem calculation, we find that our posterior distribution is

$$P(p|h) \propto p^{h+4}(1-p)^{N-h+4}$$

and relying again on the Beta function for normalization we have

$$P(p|h) = \frac{1}{B(h+5, N-h+5)}p^{h+4}(1-p)^{N-h+4}$$

Here the maximum a posterior estimate for p is $h+4/N+8$ while our posterior mean is

$$\frac{B(h+6, N-h+5)}{B(h+5, N-h+5)} = \frac{h+5}{N+10}.$$

In the situation of 55 heads out of 100, the maximum a posteriori estimate is .546 and the posterior mean is .545. These numbers have been pulled just a bit towards .5 because our prior knowledge makes us a little bit biased towards $p = .5$.

3.7.3 Bayesian Regression (or Ridge Regression)

In this chapter we return to our discussion of linear regression and introduce some Bayesian ideas. The combination will lead us to the notion of “ridge” regression, which is a type of linear regression that includes a prior distribution on the coefficients that indicates our preference for smaller rather than larger coefficients. Introduction of this prior leads to a form

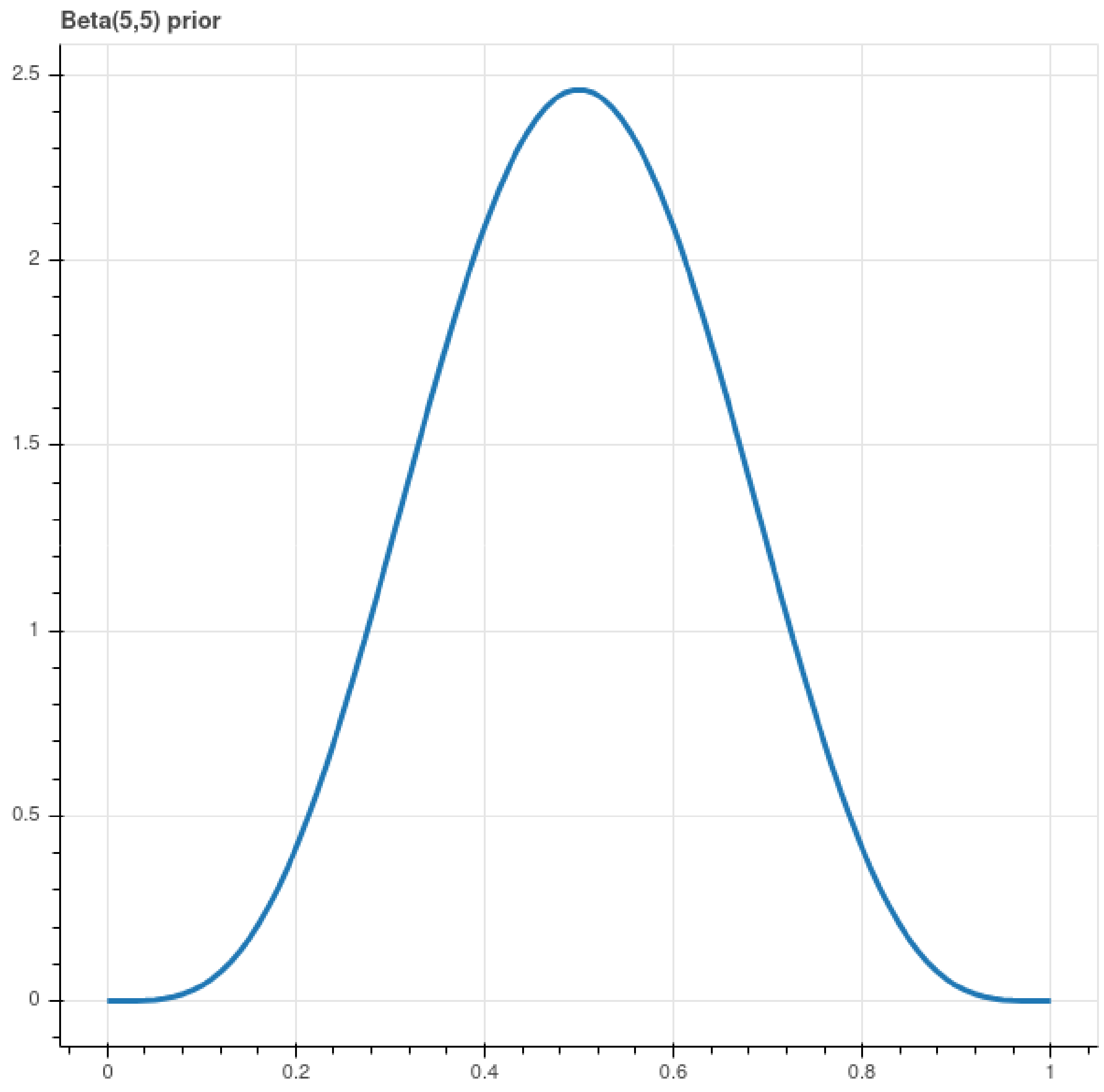


Figure 3.9: Beta(5,5) Prior

of linear regression that is more resilient in situations where the independent variables are less independent than we would hope.

Before introducing these Bayesian ideas, let us recall from Section 3.6.3 that ordinary least squares yields the parameters that give the "most likely" set of parameters for a model of the form

$$Y = XM + \epsilon$$

where the error ϵ is normally distributed with mean 0 and variance σ^2, and the mean squared error becomes the maximum likelihood estimate of the variance σ^2.

To put this into a Bayesian perspective, we notice that the linear regression model views $Y - XM$ as normally distributed given M. That is, we see the probability $P(Y - XM|M)$ as normal with variance σ^2.

Then we introduce a prior distribution on the coefficients M, assuming that they, too, are normally distributed around zero with variance τ^2. This means that *ab initio* we think that the coefficients are likely to be small.

From Bayes Theorem, we then have

$$P(M|Y, X) = \frac{P(Y, X|M)P(M)}{P(Y, X)}$$

and in distribution terms we have

$$P(M|Y, X) = Ae^{\|Y-XM\|^2/\sigma^2}e^{-\|M\|^2/\tau^2}$$

where A is a normalizing constant.

The first thing to note from this expression is that the posterior distribution for the M parameters for regression are themselves normally distributed.

The maximum likelihood estimate M_r for the parameters M occurs when $P(M|Y, X)$ is maximum, which we find by taking the derivatives. Using the matrix algebra developed in our linear regression chapter, we obtain the equation

$$(X^\top Y - (X^\top X)M_r)/\sigma^2 - M_r/\tau^2 = 0$$

or

$$(X^\top X + s)M_r = X^\top Y \tag{3.4}$$

where $s = \sigma^2/\tau^2$.

Therefore the ridge coefficients are given by the equation

$$M_r = (X^\top X + s)^{-1}X^\top Y \tag{3.5}$$

3.7.3.1 Practical aspects of ridge regression

Using ridge regression leads to a solution to the least squares problem in which the regression coefficients are biased towards being smaller. Beyond this, there are a number of implications of the technique which affect its use in practice.

First, we can put the Bayesian derivation of the ridge regression formulae in the background and focus our attention on Equation 3.5. We can treat the parameter s (which must be non-negative) as "adjustable".

One important consideration when using ridge regression is that Equation 3.5 is not invariant if we scale X and Y by a constant. This is different from "plain" regression where we consider the equation $Y = XM$. In that case, rescaling X and Y by the same factor leaves the coefficients M alone. For this reason, ridge regression is typically used on centered, standardized coordinates. In other words, we replace each feature x_i by $(x_i - \mu_i)/\sigma_i$ where μ_i and σ_i are the sample mean and standard deviation of the i^{th} feature, and we replace our response variables y_i similarly by $(y - \mu)/\sigma$ where μ and σ are the mean and standard deviation of the y-values. Then we find M using Equation 3.5, perhaps experimenting with different values of s, using our centered and standardized variables.

To emphasize that we are using centered coordinates, we write X_0 for our data matrix instead of X. Recall that the matrix $X_0^\top X_0$ that enters into Equation 3.4 is ND_0 where D_0 is the covariance matrix. Therefore in ridge regression we have replaced ND_0 by $ND_0 + s$. Since D_0 is a real symmetric matrix, as we've seen in Chapter 2 it is diagonalizable so that AD_0A^{-1} is diagonal for an orthogonal matrix A and has eigenvalues $\lambda_1 \geq \ldots \geq \lambda_k$ which are the variances of the data along the principal directions.

One effect of using ridge regression is that the eigenvalues of $ND_0 + s$ are always at least $s > 0$, so the use of ridge regression avoids the possibility that D_0 might not be invertible. In fact, a bit more is true. Numerical analysis tells us that when considering the problem of computing the inverse of a matrix, we should look at its *condition number*, which is the ratio λ_1/λ_k of the largest to the smallest eigenvalue.

If the condition number of a matrix is large, then results from numerical analysis show that it is *almost singular* and its inverse becomes very sensitive to small changes in the entries of the matrix. However, the eigenvalues of $ND_0 + s$ are $N\lambda_i + s$ and so the condition number becomes $(N\lambda_1 + s)/(N\lambda_k + s)$. For larger values of λ, this condition number shrinks, and so the inverse of the matrix $ND_0 + s$ becomes better behaved than ND_0. In this way, ridge regression helps to improve the numerical stability of the linear regression algorithm.

A second way to look at Ridge regression is to go back to the discussion of the singular value decomposition of the matrix X_0 in section Section 2.4.2. There we showed that the SVD of X_0 yields an expression

$$X_0 = U\tilde{\Lambda}P^\top$$

where U and P are orthogonal matrices and Λ is an $N \times k$ matrix whose upper block is diagonal with eigenvalues $\sqrt{N\lambda_i}$. The rows of U gave us an orthonormal basis that allowed us to write the predicted vector $\hat{Y}$ as a projection:

$$\hat{Y} = \sum_{i=1}^{k} (u_j \cdot Y) u_j^\top.$$

If we repeat this calculation, but using the ridge regression formula, we obtain

$$\hat{Y}_r = X_0 M_r = U\tilde{\Lambda}P^\top (P\tilde{\Lambda}^\top U^\top U \tilde{\Lambda} P^\top + s)^{-1} P\tilde{\Lambda}^\top U^\top Y.$$

Since P is orthogonal, $P^\top = P^{-1}$, so

$$P^\top (P\tilde{\Lambda}^2 P^\top + s)^{-1} P = P^{-1}(P(\Lambda + s)P^{-1})P = (\Lambda + s)^{-1}$$

and $\Lambda + s$ is a $k \times k$ diagonal matrix with entries $N\lambda_i + s$.

Putting the pieces together we see that

$$\hat{Y}_r = U\tilde{\Lambda}(\Lambda + s)^{-1}\tilde{\Lambda}U^\top Y.$$

In the language of orthogonal projection, this means that

$$\hat{Y}_r = \sum_{i=1}^{k} \frac{N\lambda_i}{N\lambda_i + s} (u_j \cdot Y) u_j^\top.$$

In other words, the predicted value computed by ridge regression is obtained by projecting Y into the space spanned by the feature vectors, but weighting the different principal components by $N\lambda_i/(N\lambda_i + s)$. With this weighting, the principal components with smaller variances are weighted less than those with larger variances. For this reason, ridge regression is sometimes called a *shrinkage* method.

4 The Naive Bayes classification method

4.1 Introduction

In our discussion of Bayes Theorem, we looked at a situation in which we had a population consisting of people infected with COVID-19 and people not infected, and we had a test that we could apply to determine which class an individual belonged to. Because our test was not 100 percent reliable, a positive test result didn't guarantee that a person was infected, and we used Bayes Theorem to evaluate how to interpret the positive test result. More specifically, our information about the test performance gave us the the conditional probabilities of positive and negative test results given infection status – so for example we were given $P(+|\text{infected})$, the chance of getting a positive test assuming the person is infected – and we used Bayes Theorem to determine $P(\text{infected}|+)$, the chance that a person was infected given a positive test result.

The Naive Bayes classification method is a generalization of this idea. We have data that belongs to one of two classes, and based on the results of a series of tests, we wish to decide which class a particular data point belongs to. For one example, we are given a collection of product reviews from a website and we wish to classify those reviews as either "positive" or "negative." This type of problem is called "sentiment analysis." For another, related example, we have a collection of emails or text messages and we wish to label those that are likely "spam" emails. In both of these examples, the "test" that we will apply is to look for the appearance or absence of certain key words that make the text more or less likely to belong to a certain class. For example, we might find that a movie review that contains the word "great" is more likely to be positive than negative, while a review that contains the word "boring" is more likely to be negative.

The reason for the word "naive" in the name of this method is that we will derive our probabilities from empirical data, rather than from any deeper theory. For example, to find the probability that a negative movie review contains the word "boring", we will look at a bunch of reviews that our experts have said are negative, and compute the proportion of those that contain the word boring. Indeed, to develop our family of tests, we will rely on a training set of already classified data from which we can determine estimates of probabilities that we need.

4.2 An example dataset

To illustrate the Naive Bayes algorithm, we will work with the “Sentiment Labelled Sentences Data Set” ([3]). This dataset contains 3 files, each containing 1000 documents labelled 0 or 1 for “negative” or “positive” sentiment. There are 500 of each type of document in each file. One file contains reviews of products from amazon.com; one contains yelp restaurant reviews, and one contains movie reviews from imdb.com.

Let’s focus on the amazon reviews data. Here are some samples:

```
So there is no way for me to plug it in here
    in the US unless I go by a converter.   0
Good case, Excellent value. 1
Great for the jawbone.  1
Tied to charger for conversations lasting more than
    45 minutes.MAJOR PROBLEMS!! 0
The mic is great.   1
I have to jiggle the plug to get it to line up right to
    get decent volume.  0
If you have several dozen or several hundred contacts, then
    imagine the fun of sending each of them one by one. 0
If you are Razr owner...you must have this! 1
Needless to say, I wasted my money. 0
What a waste of money and time!.    0
```

As you can see, each line consists of a product review followed by a 0 or 1; in this file the review is separated from the text by a tab character.

4.3 Bernoulli tests

We will describe the “Bernoulli” version of a Naive Bayes classifier for this data. The building block of this method is a test based on a single word. For example, let’s consider the word **great** among all of our amazon reviews. It turns out that **great** occurs 5 times in negative reviews and 92 times in positive reviews among our 1000 examples. So it seems that seeing the word **great** in a review makes it more likely to be positive. The appearances of great are summarized in Table 4.1 . We write ~**great** for the case where **great** does *not* appear.

Table 4.1: Ocurrences of **great** by type of review

	+	-	total
great	92	5	97
~**great**	408	495	903
total	500	500	1000

In this data, positive and negative reviews are equally likely so $P(+) = P(-) = .5$ From this table, and Bayes Theorem, we obtain the empirical probabilities (or “naive” probabilities).

$$P(\textbf{great}|+) = \frac{92}{500} = .184$$

and

$$P(\textbf{great}) = \frac{97}{1000} = .097$$

Therefore

$$P(+|\textbf{great}) = \frac{.184}{.097}(.5) = 0.948.$$

In other words, *if* you see the word **great** in a review, there’s a 95% chance that the review is positive.

What if you *do not* see the word **great**? A similar calculation from the table yields

$$P(+| \sim \textbf{great}) = \frac{408}{903} = .452$$

In other words, *not* seeing the word great gives a little evidence that the review is negative (there’s a 55% chance it’s negative) but it’s not that conclusive.

The word **waste** is associated with negative reviews. It’s statistics are summarized in Table 4.2 .

Table 4.2: Ocurrences of **waste** by type of review

	+	-	total
waste	0	14	14
~**waste**	500	486	986
total	500	500	1000

Based on this data, the "naive" probabilities we are interested in are:

$$P(+|\mathbf{waste}) = 0$$
$$P(+|\sim\mathbf{waste}) = .51$$

In other words, if you see **waste** you definitely have a negative review, but if you don't, you're only slightly more likely to have a positive one.

What about combining these two tests? Or using even more words? We could analyze our data to count cases in which both **great** and **waste** occur, in which only one occurs, or in which neither occurs, within the two different categories of reviews, and then use those counts to estimate empirical probabilities of the joint events. But while this might be feasible with two words, if we want to use many words, the number of combinations quickly becomes huge. So instead, we make a basic, and probably false, assumption, but one that makes a simple analysis possible.

Assumption: We assume that the presence or absence of the words **great** and **waste** in a particular review (positive or negative) are independent events. More generally, given a collection of words $w_1, \dots, w_k$, we assume that their occurences in a given review are independent events.

Independence means that we have

$$\begin{aligned} P(\mathbf{great}, \mathbf{waste}|\pm) &= P(\mathbf{great}|\pm)P(\mathbf{waste}|\pm) \\ P(\mathbf{great}, \sim\mathbf{waste}|\pm) &= P(\mathbf{great}|\pm)P(\sim\mathbf{waste}|\pm) \\ &\vdots \end{aligned}$$

So for example, if a document contains the word **great** and does *not* contain the word **waste**, then the probability of it being a positive review is:

$$P(+|\mathbf{great}, \sim\mathbf{waste}) = \frac{P(\mathbf{great}|+)P(\sim\mathbf{waste}|+)P(+)}{P(\mathbf{great}, \sim\mathbf{waste})}$$

while the probability of it being a negative review is

$$P(-|\mathbf{great}, \sim\mathbf{waste}) = \frac{P(\mathbf{great}|-)P(\sim\mathbf{waste}|-)P(-)}{P(\mathbf{great}, \sim\mathbf{waste})}$$

Rather than compute these probabilities (which involves working out the denominators), let's just compare them. Since they have the same denominators, we just need to compare numerators, which we call L for likelihood: Using the data from Table 4.1 and Table 4.2 , we obtain:

$$L(+|\mathbf{great}, \sim\mathbf{waste}) = (.184)(1)(.5) = .092$$

and

$$L(-|\textbf{great}, \sim \textbf{waste}) = (.01)(.028)(.5) = .00014$$

so our data suggests strongly that this is a positive review.

4.4 Feature vectors

To generalize this, suppose that we have extracted keywords $w_1, \ldots, w_k$ from our data and we have computed the empirical values $P(w_i|+)$ and $P(w_i|-)$ by counting the fraction of positive and negative reviews that contain the word w_i:

$$P(w_i|\pm) = \frac{\text{number of } \pm \text{ reviews that mention } w_i}{\text{number of } \pm \text{ reviews total}}$$

Notice that we only count *reviews*, not *ocurrences*, so that if a word occurs multiple times in a review it only contributes 1 to the count. That's why this is called the *Bernoulli* Naive Bayes – we are thinking of each keyword as yielding a yes/no test on each review.

Given a review, we look to see whether each of our k keywords appears or does not. We encode this information as a vector of length k containing 0's and 1's indicating the absence or presence of the kth keyword. Let's call this vector the *feature vector* for the review.

For example, if our keywords are $w_1 = \textbf{waste}$, $w_2 = \textbf{great}$, and $w_3 = \textbf{useless}$, and our review says

```
This phone is useless, useless, useless!  What a waste!
```

then the associated feature vector is $f = (1, 0, 1)$.

For the purposes of classification of our reviews, we are going to forget entirely about the text of our reviews and work only with the feature vectors. From an abstract perspective, then, by choosing our k keywords, our "training set" of N labelled reviews can be replaced by an $N \times k$ matrix $X = (x_{ij})$ with entries 0 or 1, where $x_{ij} = 1$ if and only if the j^{th} keyword appears in the i^{th} review.

The labels of 0 or 1 for unfavorable or favorable reviews can also be packaged up into a $N \times 1$ vector Y that serves as our "target" variable.

Setting things up this way lets us express the computations of our probabilities $P(w_i|\pm)$ in vector form. In fact, $Y^\top X$ is the sum of the rows of X corresponding to positive reviews, and therefore, letting $N_\pm$ denote the number of $\pm$ reviews,

$$P_+ = \frac{1}{N_+} Y^\top X = \begin{bmatrix} P(w_1|+) & P(w_2|+) & \cdots & P(w_k|+) \end{bmatrix}.$$

Similarly, since Y and X have zero and one entries only, if we write $1-Y$ and $1-X$ for the matrices obtained by replacing every entry z by $1-z$ in each matrix, we have:

$$P_- = \frac{1}{N_-}(1-Y)^\top X = \left[\begin{array}{cccc} P(w_1|-) & P(w_2|-) & \cdots & P(w_k|-) \end{array}\right].$$

Note that the number of positive reviews is $N_+ = Y^\top Y$ and the number of negative ones is $N_- = N - N_+$. Since $P(+)$ is the fraction of positive reviews among all reviews, we can compute it as $P(+) = \frac{1}{N}Y^\top Y$, and $P(-) = 1 - P(+)$.

4.5 Likelihood

If a review has an associated feature vector $f = (f_1, \ldots, f_k)$, then by independence the probability of that feature vector ocurring within one of the $\pm$ classes is

$$P(f|\pm) = \prod_{i:f_i=1} P(w_i|\pm) \prod_{i:f_i=0} (1 - P(w_i|\pm))$$

which we can also write

$$P(f|\pm) = \prod_{i=1}^{k} P(w_i|\pm)^{f_i}(1 - P(w_i|\pm))^{(1-f_i)}. \tag{4.1}$$

These products aren't practical to work with – they are often the product of many, many small numbers and are therefore really tiny. Therefore it's much more practical to work with their logarithms.

$$\log P(f|\pm) = \sum_{i=1}^{k} f_i \log P(w_i|\pm) + (1-f_i)\log(1 - P(w_i|\pm)) \tag{4.2}$$

If we have a group of reviews N organized in a matrix X, where each row is the feature vector associated to the corresponding review, then we can compute all of this at once. We'll write $\log P_\pm = \log P(X|\pm)$ as the row vector whose i^{th} entry is $\log P(f_i|\pm)$:

$$\log P(X|\pm) = X(\log P_\pm)^\top + (1-X)(\log(1-P_\pm))^\top. \tag{4.3}$$

By Bayes Theorem, we can express the chance that our review with feature vector f is positive or negative by the formula:

$$\log P(\pm|f) = \log P(f|\pm) + \log P(\pm) - \log P(f)$$

where

$$P(\pm) = \frac{\text{the number of } \pm \text{ reviews}}{\text{total number of reviews}}$$

and $P(f)$ is the fraction of reviews with the given feature vector. (Note: in practice, some of these probabilities will be zero, and so the log will not be defined. A common practical approach to dealing with this is to introduce a "fake document" into both classes in which every vocabulary word appears – this guarantees that the frequency matrix will have no zeros in it).

A natural classification rule would be to say that a review is positive if $\log P(+|f) > \log P(-|f)$, and negative otherwise. In applying this, we can avoid computing $P(f)$ by just comparing $\log P(f|+) + \log P(+)$ and $\log P(f|-) + \log P(-)$ computed using Equation 4.2. Then we say:

- a review is positive if $\log P(f|+) + \log P(+) > \log P(f|-) + \log P(-)$ and negative otherwise.

Again we can exploit the matrix structure to do this for a bunch of reviews at once. Using Equation 4.3 and the vectors $P_{\pm}$ we can compute column vectors corresponding to both sides of our decision inequality and subtract them. The positive entries indicate positive reviews, and the negative ones, negative reviews.

4.6 The Bag of Words

In our analysis above, we thought of the presence or absence of certain key words as a set of independent tests that provided evidence of whether our review was positive or negative. This approach is suited to short pieces of text, but what about longer documents? In that case, we might want to consider not just the presence or absence of words, but the frequency with which they appear. Multinomial Naive Bayes, based on the "bag of words" model, is a classification method similar to Bernoulli Naive Bayes but which takes term frequency into account.

Let's consider, as above, the problem of classifying documents into one of two classes. We assume that we have a set of keywords $w_1, \ldots, w_k$. For each class $\pm$, we have a set of probabilities $P(w_i|\pm)$ with the property that

$$\sum_{i=1}^{k} P(w_i|\pm) = 1.$$

The "bag of words" model says that we construct a document of length N in, say, the $+$ class by independently drawing a word N times from the set $w_1, \ldots, w_k$ with probabilities $P(w_i|+)$. The name "bag of words" comes from thinking of each class as having an associated bag containing the words $w_1, \ldots, w_k$ with relative frequencies given by the probabilities, and generating a document by repeatedly drawing a word from the bag.

In the Multinomial Naive Bayes method, we estimate the probabilities $P(w_i|\pm)$ by counting the number of times each word occurs in a document of the given class:

$$P(w_i|\pm) = \frac{\text{number of times word } i \text{ occurs in } \pm \text{ documents}}{\text{total number of words in } \pm \text{ documents}}$$

This is the "naive" part of the algorithm. Package up these probabilities in vectors:

$$P_{\pm} = \left[\begin{array}{ccc} P(w_1|\pm) & \cdots & P(w_k|\pm) \end{array} \right].$$

As in the Bernoulli case, we often add a fake document to each class where all of the words occur once, in order to avoid having zero frequencies when we take a logarithm later.

Now, given a document, we associate a feature vector $\mathbf{f}$ whose i^{th} entry is the frequency with which word i appears in that document. The probability of obtaining a particular document with feature vector $\mathbf{f} = (f_1, \ldots, f_k)$ from the bag of words associated with class $\pm$ is given by the "multinomial" distribution:

$$P(\mathbf{f}|\pm) = \frac{N!}{f_1! f_2! \cdots f_k!} \prod_{i=1}^{k} P(w_i|\pm)^{f_i}$$

which generalizes the binomial distribution to multiple choices. The constant will prove irrelevant, so let's call the interesting part $L_{\pm}$:

$$L(\mathbf{f}|\pm) = \prod_{i=1}^{k} P(w_i|\pm)^{f_i}$$

From Bayes Theorem, we have

$$P(\pm|\mathbf{f}) = \frac{P(\mathbf{f}|\pm)P(\pm)}{P(\mathbf{f})}$$

where $P(\pm)$ is estimated by the fraction of documents (total) in each class.

We classify our document by considering $P(\pm|\mathbf{f})$ and concluding:

- a document with feature vector $\mathbf{f}$ is in class $+$ if $\log P(+|\mathbf{f}) > \log P(-|\mathbf{f})$.

In this comparison, both the constant (the multinomial coefficient) and the denominator cancel out, so we only need to compare $\log L(\mathbf{f}|+) + \log P(+)$ with $\log L(\mathbf{f}|-) + \log P(-)$ We have

$$\log L(\mathbf{f}|\pm) = \sum_{i=1}^{k} f_i \log P(w_i|\pm)$$

or, in vector form,

$$\log P(\mathbf{f}|\pm) = \mathbf{f} \log P_{\pm}^{\top}$$

Therefore, just as in the Bernoulli case, we can package up our document i as an $N \times k$ data matrix X, where position ij gives the number of times word j occurs in document i. Then we can compute the vector

$$\hat{Y} = X \log P_{+}^{\top} + \log P(+) - X \log P_{-}^{\top} - \log P(-)$$

and assign those documents where $\hat{Y} > 0$ to the $+$ class and the rest to the $-$ class.

4.7 Other applications

We developed the Naive Bayes method for sentiment analysis, but once we chose a set of keywords our training data was reduced to an $N \times k$ matrix X of 0/1 entries, together with an $N \times 1$ target column vector Y. Then our classification problem is to decide whether a given vector of k entries, all 0 or 1, is more likely to carry a 0 or 1 label. All of the parameters we needed for Naive Bayes – the various probabilities – can be extracted from the matrix X.

For example, suppose we have a collection of images represented as black/white pixels in a grid that belong to one of two classes. For example, we might have $28x28$ bitmaps of handwritten zeros and ones that are labelled, and we wish to construct a classifier that can decide whether a new $28x28$ bitmap is a zero or one. An example of such a bitmap is given in Figure 4.1. We can view each $28x28$ bitmap as a vector of length 784 with 0/1 entries and apply the same approach outlined above. However, there are other methods that are more commonly used for this problem, such as logistic regression and neural networks.

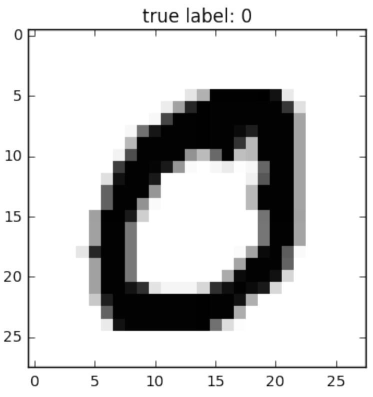


Figure 4.1: Handwritten 0

5 Logistic Regression

Suppose that we are trying to convince customers to buy our product by showing them advertising. Our experience teaches us that there is no deterministic relationship between how often a potential customer sees one of our ads and whether or not they purchase our product, nevertheless it is the case that as they see more ads they become more likely to make a purchase. Logistic regression is a statistical model that can capture the essence of this idea.

To make this problem more abstract, let's imagine that we are trying to model a random event that depends on a parameter. As in our introduction above, the random event might be a user deciding to make a purchase from a website, which, in our very simple model, depends on how many times the user saw an advertisement for the product in question. But we could imagine other situations where the chance of an event happening depends on a parameter. For example, we could imagine that a student's score on a certain test depends on how much studying they do, with the likelihood of passing the test increasing with the amount of studying.

To construct this model, we assume that the probability of a certain event p is related to some parameter x by the following relationship:

$$\log \frac{p}{1-p} = ax + b \tag{5.1}$$

where a and b are constants. The quantity $\frac{p}{1-p}$ is the "odds" of the event occurring. We often use this quantity colloquially; if the chance of our team winning a football game is 1 in 3, then we would say the odds of a win are 1-to-2, which we can interpret as meaning they are twice as likely to lose as to win. The quantity $\log \frac{p}{1-p}$ is, for obvious reasons, called the log-odds of the event.

The assumption in Equation 5.1 can be written

$$\frac{p}{1-p} = e^{ax+b}$$

and we interpret this as telling us that if the parameter x increases by 1, the odds of our event happening go up by a factor of e^a. So, to be even more concrete, if $a = \log 2$, then our logistic model would say that an increase of 1 in our parameter x doubles the odds of our event taking place.

In terms of the probability p, Equation 5.1 can be rewritten

$$p = \frac{1}{1 + e^{-ax-b}}$$

This proposed relationship between the probability p and the parameter x is called the *logistic model.* The function

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

is called the *logistic function* and yields an S-shaped curve.

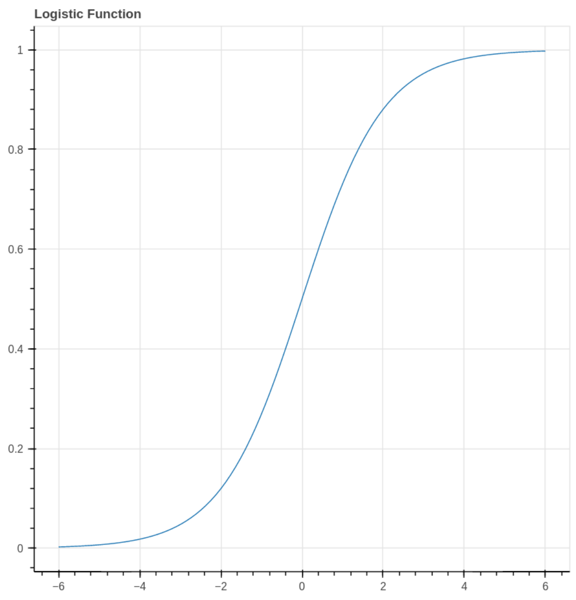


Figure 5.1: Logistic Curve

To fully put the logistic model in perspective, let's choose some explicit parameters and look at what data arising from such a model would look like. Imagine therefore that $a = \log 2$ and $b = 0$, so that the probability of the event we are interested occurring is given by the formula

$$p(x) = \frac{1}{1 + e^{-(\log 2)x}} = \frac{1}{1 + (.5)^x}.$$

Our data consists of counts of how often our event happened for a range of values of x. To generate this data, we can pick x values from the set $\{-3, -2, -1, 0, 1, 2, 3\}$ yielding probabilities $\{.11, .2, .33, .4, .56, .67, .8\}$. Now our data consists of, for each value of x, the result of 100 independent Bernoulli trials with probability $p(x)$. For example, we might find that our event occurred $\{10, 18, 38, 50, 69, 78, 86\}$ times respectively for each of the x values.

5.1 Likelihood and Logistic Regression

In applications, our goal is to choose the parameters of a logistic model to accurately predict the likelihood of the event under study occurring as a function of the measured parameter. Let's imagine that we collected the data that we generated above, without knowing that it's source was a logistic model. So Table 5.1 shows the number of times the event occurred, for each of the measured values of the x parameter.

Table 5.1: Sample Data

x	-3	-2	-1	0	1	2	3
Occurrences (out of 100)	10	18	38	50	69	78	86

Our objective now is to find a logistic model which best explains this data. Concretely, we need to estimate the coefficients a and b that yield

$$p(x) = \frac{1}{1+e^{-ax-b}} \tag{5.2}$$

where the resulting probabilities best estimate the data. As we have seen, this notion of "best" can have different interpretations. For example, we could approach this from a Bayesian point of view, adopt a prior distribution on the parameters a and b, and use the data to obtain this prior and obtain a posterior distribution on a and b. For this first look at logistic regression, we will instead adopt a "maximum likelihood" notion of "best" and ask what is the most likely choice of a and b to yield this data.

To apply the maximum likelihood approach, we need to ask "for (fixed, but unknown) values of a and b, what is the likelihood that a logistic model with those parameters would yield the data we have collected?" Each column in Table 5.1 represents 100 Bernoulli trials with a fixed probability $p(x)$. So, for example, the chance q of obtaining 10 positive results with $x=-3$ is given by

$$q(-3) = Cp(-3)^{10}(1-p(-3))^{90}$$

where C is a constant (it would be a binomial coefficient). Combining this for different values of x, we see that the likelihood of the data is the product

$$L(a,b) = C'p(-3)^{10}(1-p(-3))^{90}p(-2)^{18}(1-p(-2))^{82}\cdots p(3)^{86}(1-p(3))^{14}$$

where C' is another constant. Each $p(x)$ is a function of the parameters a and b, so all together this is a function of those two parameters. Our goal is to maximize it.

One step that simplifies matters is to consider the logarithm of the likelihood:

$$\log L(a,b) = \sum_{i=0}^{6} \left[x_i \log(p(x_i)) + (100-x_i)\log(1-p(x_i))\right] + C''$$

where C'' is yet another constant. Since our ultimate goal is to maximize this, the value of C'' is irrelevant and we can drop it.

5.1.1 Another point of view on logistic regression

In Table 5.1 we summarize the results of our experiments in groups by the value of the x parameter. We can think of the data somewhat differently, by instead considering each event separately, corresponding to a parameter value x and an outcome 0 or 1. From this point of view the data summarized in Table 5.1 would correspond to a vector with 700 rows. The first 100 rows (corresponding to the first column of the table) would have first entry -3, the next 100 would have -2, or so on. So our parameter values form a vector X. Meanwhile, the outcomes form a vector Y with entries 0 or 1.

More generally, imagine we are studying our advertising data and, for each potential customer, we record how many times they saw our ad. We create a vector X whose entries are these numbers. Then we create another vector Y, of the same length, whose entries are either 0 or 1 depending of whether or not the customer purchased our product.

One way to think about logistic regression in this setting is that we are trying to fit a function that, given the value x_i, tries to yield the corresponding value y_i. However, instead of finding a deterministic function, as we did in linear regression, instead we try to fit a logistic function that captures the likelihood that the y-value is a 1 given the x-value. This "curve-fitting perspective" is why this is considered a regression problem.

If, as above, we think of each row of the matrix as an independent trial, then the chance that $y_i = 1$ is $p(x_i)$ and the chance that $y_i = 0$ is $1 - p(x_i)$, where $p(x)$ is given by the logistic function as in Equation 5.2. The likelihood of the results we obtained is therefore:

$$L(a, b) = C \prod_{i=0}^{N-1} p(x_i)^{y_i}(1 - p(x_i))^{(1-y_i)}$$

where C is a constant and we are exploiting the trick that, since y_i is either zero or one, $1 - y_i$ is correspondingly one or zero. Thus only $p(x_i)$ or $1 - p(x_i)$ occurs in each term of the product. If we group the terms according to x_i we obtain our earlier formula for $L(a, b)$.

This expresssion yields an apparently similar formula for the log-likelihood (up to an irrelevant constant):

$$\log L(X, a, b) = \sum_{i=0}^{N-1} y_i \log p(x_i) + (1 - y_i) \log(1 - p(x_i)).$$

Using vector notation, this can be further simplified, where again we drop irrelevant constants:

$$\log L(X, a, b) = Y \cdot \log p(X) + (1 - Y) \cdot \log(1 - p(X)).$$

To be absolutely concrete, in this formula, $p(X)$ is a vector

$$p(X) = [p(x_i)]_{i=0}^{N-1} = \left[\frac{1}{1+e^{-ax_i-b}}\right]_{i=0}^{N-1}$$

so its entries are functions of the unknown parameters a and b.

We might naively try to maximize this by taking the derivatives with respect to a and b and setting them to zero, but this turns out to be impractical. So we need a different approach to finding the parameters a and b which maximize this likelihood function. We will return to this problem later, but before we do so we will look at some generalizations and broader applications of the logistic model.

5.1.2 Logistic regression with multiple features

The next generalization we can consider of the logistic model is the situation where the log-odds of our event of interest depend linearly on multiple parameters. In other words, we have

$$\log \frac{p}{1-p} = m_0x_0 + m_1x_1 + \cdots + m_{k-1}x_{k-1} + b$$

where the a_i and b are constants. Under this model, notice that *the incremental effects of changes to the different parameters x_i have independent effects on the probability.* So, for example, if x_1 were the number of times our potential customer saw an online advertisement and x_2 were the number of times they saw a print advertisement, by adopting this model we are assuming that the impact of seeing more online ads is completely unrelated to the impact of seeing more print ads.

The probability is again given by a sigmoid function

$$p(x_1, \ldots, x_k) = \frac{1}{1+e^{-\sum_{i=0}^{k-1} m_ix_i+b}}$$

This model has an $N \times k$ feature matrix whose rows are the values $x_0, \ldots, x_{k-1}$ for each sample. The outcome of our experimemt is recorded in an $N \times 1$ column vector Y whose entries are 0 or 1. The likelihood function is formally equivalent to what we computed in the case of a single feature, but it will be useful to be a bit careful about vector notation.

Following the same pattern we adopted for linear regression, let X be the $N \times (k+1)$ matrix whose first k columns contain the values x_i for each sample, and whose last column is all 1. Rename the "intercept" variable as a_{k+1} and organize these parameters into a $(k+1) \times 1$ matrix M. Then

$$p(X) = \sigma(XM)$$

and our likelihood becomes

$$\log L(M) = Y \cdot \log \sigma(XM) + (1-Y) \cdot (1 - \log \sigma(XM)). \tag{5.3}$$

5.2 Finding the maximum likelihood solution by gradient descent

Given a set of features X and targets Y for a logistic model, we now want to find the values M so that the log-likelihood of the model for those paramters, given the data, is maximized. While in linear regression we could find a nice closed form solution to this problem, the presence of the non-linear function $\sigma(x)$ in the likelihood makes that impossible for logistic regression. Thus we need to use a numerical approximation. The most straightforward such method is called gradient descent. It is at the foundation of many numerical optimization algorithms, and so while we will develop it here for logistic regression we will have other opportunities to apply it and we will discuss it more thoroughly on its own later.

5.2.1 Gradient descent

Suppose that we have a function $f(x_0, \dots, x_{k-1})$ and we wish to find its minimum value. To apply gradient descent, we choose an initial starting point $c = (c_0, \dots, c_{k-1})$ and we iteratively adjust the values of c so that the values $f(c)$ decrease. When we can no longer do that, we've found what is at least a local minimum of f.

How should we make these adjustments? Let us remember the idea of the *directional derivative* from multivariate calculus. The directional derivative $D_v f$ measures the rate of change of f as one moves with velocity vector v from the point x and it is defined as

$$D_v f(x) = \frac{d}{dt} f(x + tv)|_{t=0}$$

From the chain rule, we can compute that

$$D_v f(x) = \sum_{i=0}^{k-1} \frac{\partial f}{\partial x_i} \frac{dx_i}{dt} = (\nabla f) \cdot v$$

where

$$\nabla f = \left[\frac{\partial f}{\partial x_i} \right]_{i=0}^{k-1}$$

is the gradient of f. This argument yields the following result.

Proposition: Let $f(x_0, \dots, x_{k-1})$ be a smooth function from $\mathbb{R}^k \to \mathbb{R}$. Then for every point $c = (c_0, \dots, c_{k-1})$, if $\nabla f \neq 0$ then ∇f is a vector pointing in the direction in which f increases most rapidly, and $-\nabla f$ is a vector pointing in the direction in which f decreases most rapidly. If $\nabla f = 0$, then c is a critical point of f.

Proof: The directional derivative $D_v(f) = (\nabla f) \cdot v$ measures the rate of change of f if we travel with velocity v from a point x. To remove the dependence on the magnitude of v (since obviously f will change more quickly if we travel more quickly in a given direction), we scale

v to be a unit vector. Then the dot product giving the rate is maximized when v is parallel to ∇f.

This observation about the gradient yields the algorithm for gradient descent.

Gradient Descent Algorithm: Given a function $f : \mathbb{R}^k \to \mathbb{R}$, choose a point $c^{(0)}$, a small constant ν (called the *learning rate*) and a small constant ϵ (the *tolerance*). Then iteratively compute

$$c^{(n+1)} = c^{(n)} - \nu \nabla f(c^{(n)})$$

until $|c^{(n+1)} - c^{(n)}| < \epsilon$. Then $c^{(n+1)}$ is an (approximate) critical point of f.

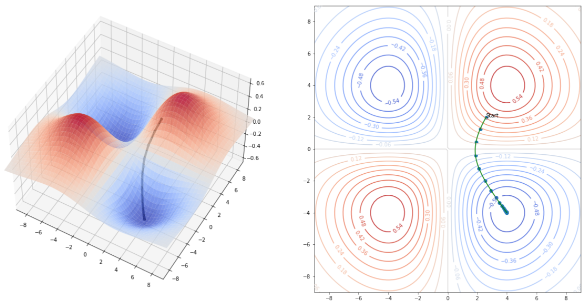


Figure 5.2: Gradient Descent

The behavior of gradient descent is illustrated in Figure 5.2 for the function

$$f(x, y) = \frac{xy}{\sigma\sqrt{2\pi}} e^{(-x^2-y^2)/2\sigma^2}$$

where $\sigma = 4$. This function has two "upward" humps and two "downward" humps. Starting on the inside slope of one of the upward humps, gradient descent finds the bottom of an adjacent "downward" hump.

To get a little more perspective on gradient descent, consider the one-dimensional case, with $f(x) = 4x^3 - 6x^2$. This is a cubic polynomial whose graph has a local maximum and a local minimum, depicted in Figure 5.3.

In this case the gradient is just the derivative $f'(x) = 12x^2 - 12x$ and the iteration is

$$c^{(n+1)} = c^{(n)} - 12\nu((c^{(n)})^2 - c^{(n)}).$$

Even from this simple example we can see the power and also the pitfalls of this method. Suppose we choose $x_0 = 2$, $\nu = .01$, and $\epsilon = .001$. Then the iteration yields:

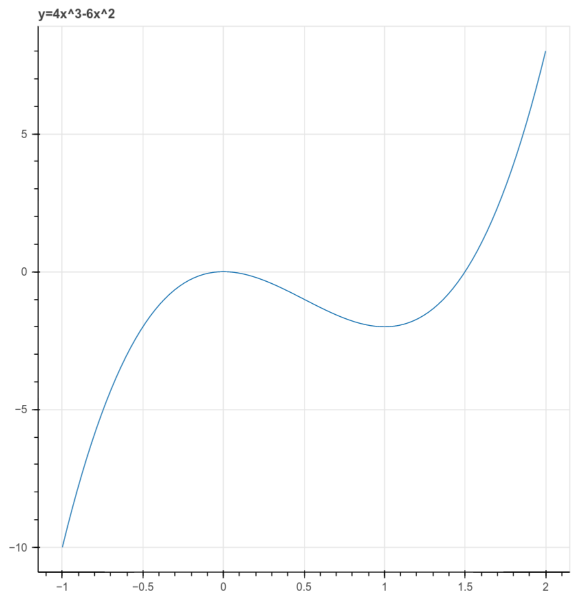


Figure 5.3: A cubic polynomial

Table 5.2: Gradient Descent Iterations

Step	0	1	2	3	4	5	6
x	2.0	0.8	0.896	0.952	0.979	0.991	0.997

As you can see, the points move quickly to the (local) minimum at $x = 1$.

There are two ways (at least) that things can go wrong, however. First suppose we use $x_0 = -1$, instead of $x_0 = 2$, as our first guess. Then we are on the downslope on the left side of the graph, and following the gradient quickly takes us off to $-\infty$.

Table 5.3: Gradient Descent Iterations (first failure mode)

Step	0	1	2	3	4	5
x	-1.00	-2.20	-6.42	-35.04	-792.70	-378296.27

Second, suppose we choose $x_0 = 2$, but choose a somewhat larger learning rate – say, $\nu = .1$. In this case, initially things look good, but the addition of the gradient causes an overshoot which once again takes us over the hump at $x = 0$ and off to $-\infty$ heading to the left.

Table 5.4: Gradient Descent Iterations (second failure mode)

Step	0	1	2	3	4	5	6
x	2.00	-0.11	-0.24	-0.56	-1.49	-5.42	-42.23

Based on these considerations, we see that, for general functions, *if gradient descent converges,* then it will converge to a local minimum of the function. But *it may not converge,* and even if it does, we can't conclude anything about whether we've reached a *global* minimum.

5.3 Gradient Descent and Logistic Regression

The key piece of information that we need is the gradient $-\nabla L$, where the variables are the entries of M. The complicating features is the presence of the nonlinear function σ, so let's start with a simple observation about this function.

Lemma: The logistic function $\sigma(x)$ satisfies the differential equation

$$\frac{d\sigma}{dx} = \sigma(x)(1-\sigma(x)).$$

Proof: Since

$$\sigma(x) = \frac{1}{1+e^{-x}},$$
$$1-\sigma(x) = \frac{e^{-x}}{1+e^{-x}}.$$

Then we calculate

$$\begin{aligned}\frac{d\sigma}{dx} &= \left(\frac{1}{(1+e^{-x})}\right)^2 e^{-x} \\ &= \left(\frac{1}{1+e^{-x}}\right)\left(\frac{e^{-x}}{1+e^{-x}}\right) \\ &= \sigma(x)(1-\sigma(x))\end{aligned}$$

which is what we claimed.

We apply this differential equation to compute the gradient of L.

Theorem 5.1. ***Proposition:*** *The gradient $-\nabla L(M)$ is given by*

$$-\nabla \log L(M) = X^\top(\sigma(XM) - Y).$$

Notice that the right side of this equation yields a $(k+1)\times 1$ column vector. The entries of this vector are the partial derivatives with respect to the coefficients m_i for $i = 0, \ldots, k$.

Proof: This is yet another exercise in the chain rule and keeping track of indices. Let's first look at the term $Y \cdot \log \sigma(XM)$. Writing it out, we have

$$Y \cdot \log \sigma(XM) = \sum_{i=0}^{N-1} y_i \log \sigma(\sum_{j=0}^{k} x_{ij} m_j).$$

Applying $\partial/\partial m_s$ to this yields

$$\sum_{i=0}^{N-1} y_i (1 - \sigma(\sum_{j=0}^{k} x_{ij} m_j)) x_{is}$$

where we've used the chain rule and the differential equation for σ discussed above. At the same time, we can apply $\partial/\partial m_s$ to the second term $(1-Y)\cdot\log(1-\sigma(XM))$ and obtain

$$-\sum_{i=0}^{N-1}(1-y_i)\sigma(\sum_{j=0}^{k}x_{ij}m_j)x_{is}.$$

The term $\sum_{i=0}^{N-1}y_i\sigma(\sum_{j=0}^{k}x_{ij}m_j)x_{is}$ cancels, yielding

$$\frac{\partial L(M)}{m_s}=-\sum_{i=0}^{N-1}(y_i-\sigma(\sum_{j=0}^{k}x_{ij}m_j))x_{is}.$$

Since our weights M are naturally a $(k+1)\times 1$ column vector, looked at properly this is our desired formula:

$$-\nabla\log L(M)=X^{\top}(\sigma(XM)-Y).$$

Since the right side is an $(k+1)\times N$ matrix times an $N\times 1$ column vector, the result is a $(k+1)\times 1$ column vector whose entries are the partial derivatives of $-\log L(M)$ with respect to the weights m_s.

5.3.1 Gradient Descent on our synthetic data

Now we can apply gradient descent to find a maximum likelihood logistic model for the sample data that we generated from the logistic model and reported in Table 5.1. With the probability given as

$$p(x)=\frac{1}{1+e^{-ax-b}}$$

we make an initial guess of $a=1$ and $b=0$ set a learning rate $\nu=.001$, and run the gradient descent algorithm for 30 iterations. We plot the negative log-likelihood for this algorithm one the left in Figure 5.4, where we see that it drops swiftly to a minimum value. The corresponding parameter values are $a=.6717$ and $b=-.0076$, and the fit of the the corresponding logistic curve to the observed data is shown on the right in Figure 5.4.

The parameters used to generate the data are close to this; they were $a=log(2)=.6931\$$ and $b=0$.

5.3.2 Gradient Descent and Logistic Regression on "real" data

We conclude this first look at logistic regression and gradient descent by analyzing some simple real data. This dataset consists of about 2200 customers who patronize a certain food store. Among the features in the data set is a field giving the total dollars spent at the store by a customer; we will study that feature and its relationship to the question of whether or not the customer accepted a special offer from the store. (see [4] for the original data source).

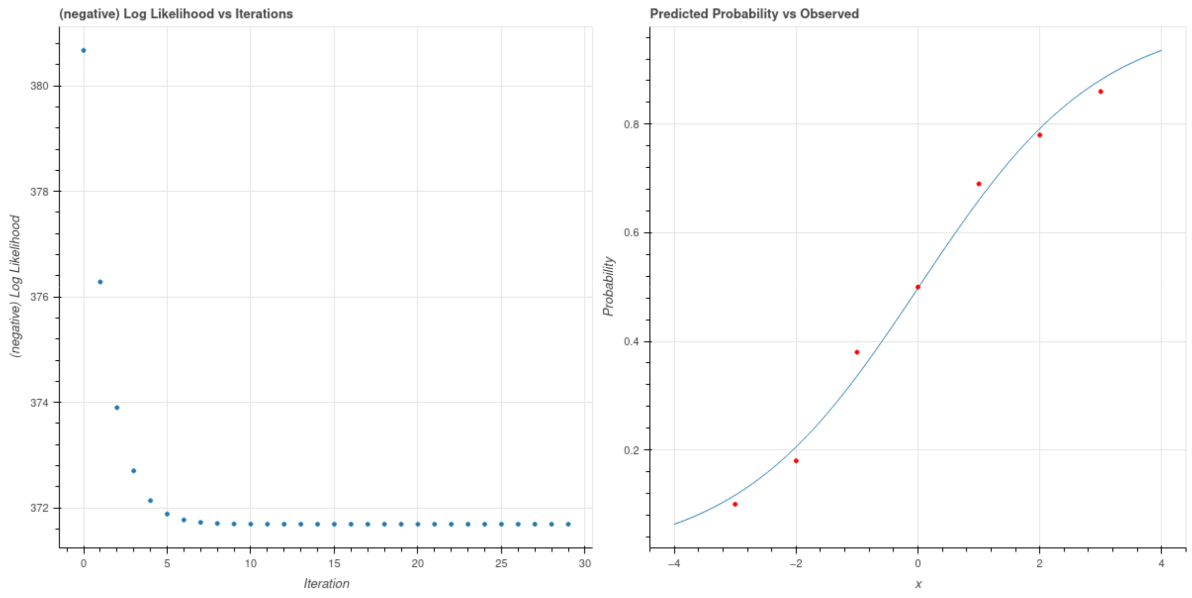


Figure 5.4: Max Likelihood Gradient Descent for Logistic Fitting

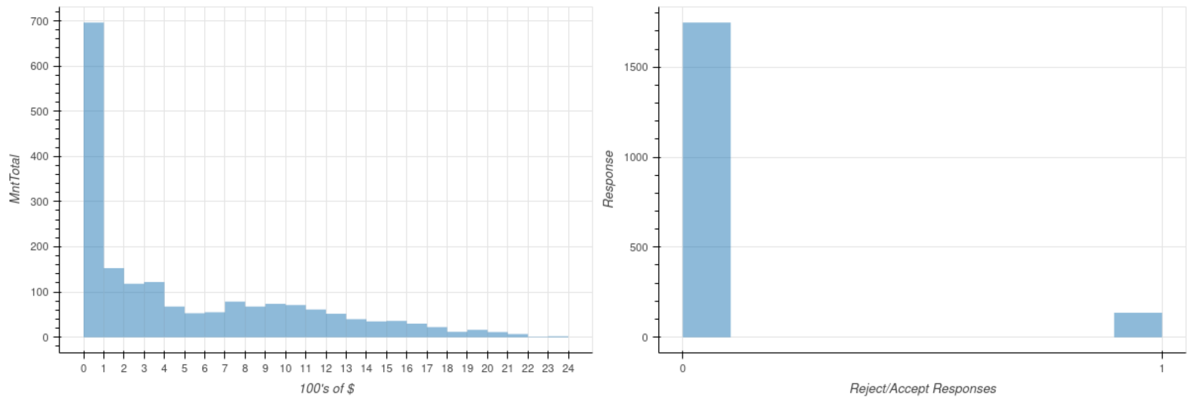


Figure 5.5: Food Marketing Data: Histograms of Expenditures and Response

The two plots in Figure 5.5 summarize the data. The first plot is a histogram showing the amounts spent by the customers; the second shows the distribution of responses.

We would like to know how expenditures increase the likelihood of customers accepting our offer. We therefore fit a logistic model to the data. The result is shown in Figure 5.6.

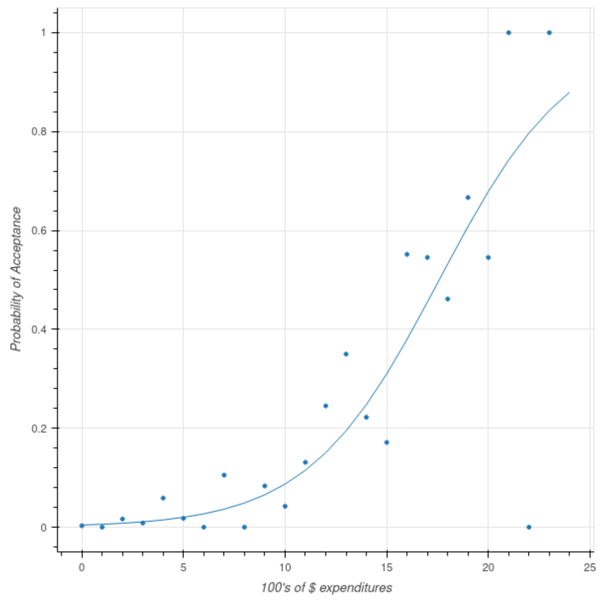


Figure 5.6: Logistic Model for Food Marketing

5.4 Logistic Regression and classification

Beyond the kind of probability prediction that we have discussed up to this point, logistic regression is one of the most powerful techniques for attacking the classification problem. Let's start our discussion with a sample problem that is a simplified version of one of the most famous machine learning benchmark problems, the MNIST (Modified National Institute of Science and Technology) dataset of handwritten numerals. This dataset consists of 60000 labelled grayscale images of handwritten digits from 0 to 9. Each image is stored as a $28x28$ array of integers from 0 to 255. Each cell of the array corresponds to a "pixel" in the image, and the contents of that cell is a grayscale value. See [5] for the a more detailed description of how the dataset was constructed.

In Figure 5.7 is a picture of a handwritten "1" from the MNIST dataset.

Classification Problem for MNIST: Given a $28x28$ array of grayscale values, determine which digit is represented.

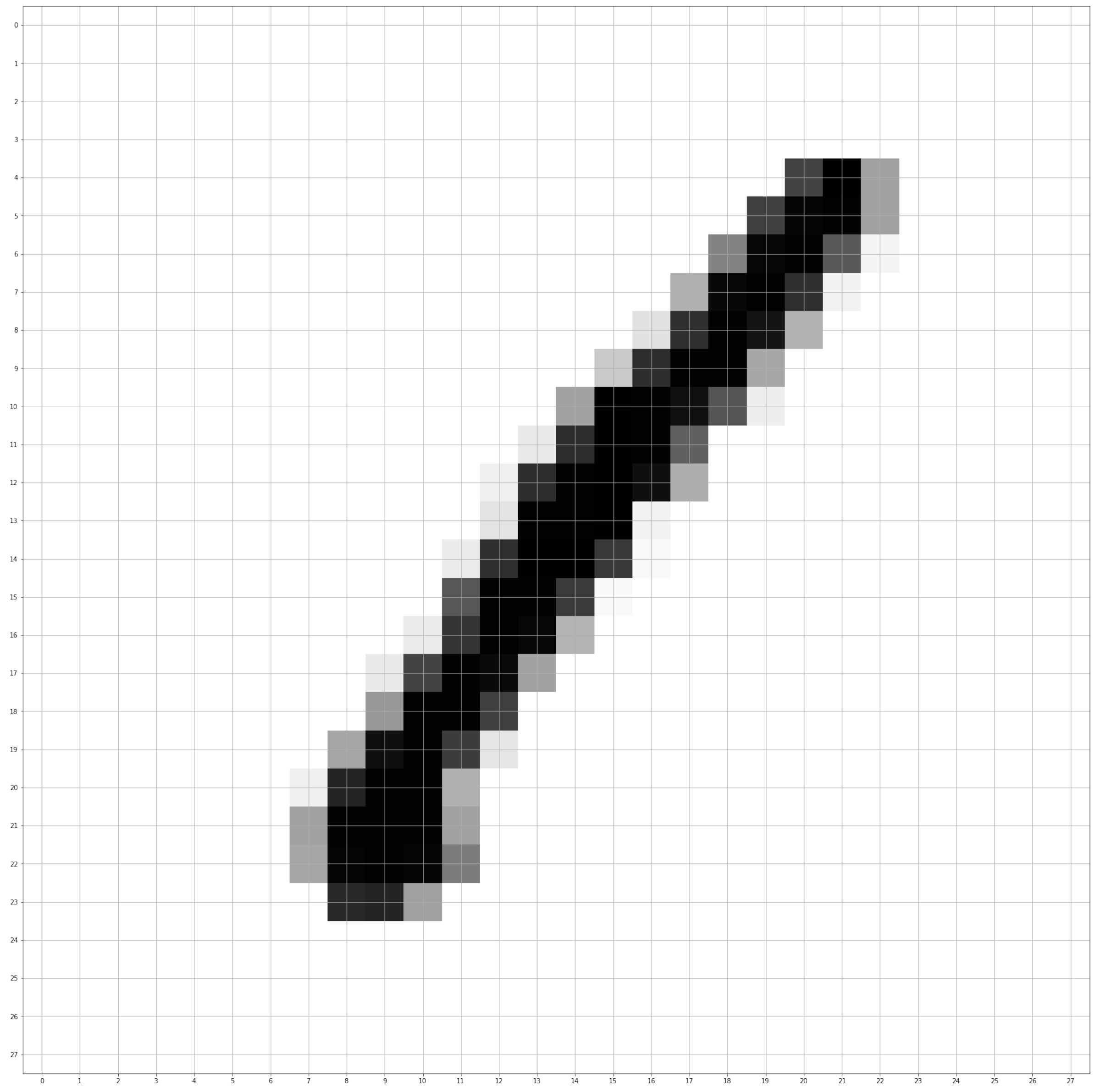


Figure 5.7: Handwritten One from MNIST

At first glance, this does not look like a logistic regression problem. To make the connection clearer, let's simplify the problem and imagine that our database contains only labelled images of zeros and ones – we'll worry about how to handle the full problem later. So now our task is to determine which images are zeros, and which are ones.

Our approach will be to view each image as a vector of length $784 = 28 * 28$ by stringing the pixel values row by row into a one dimensional vector, which following our conventions yields a matrix of size $N \times 784$ where N is the number of images. Since we may also need an "intercept", we add a column of 1's to our images yielding a data matrix X of size $N \times 785$. The labels y form a column vector of size N containing zeros and ones.

We will also simplify the data but converting the gray-scale images to monochrome by converting gray levels up to 128 as "white" and beyond 128 as "black".

The logistic regression approach asks us to find the "best" vector M so that, for a given image vector x (extended by adding a one at the end), the function

$$p(x) = \frac{1}{1 + e^{-xM}}$$

is close to 1 if x represents a one, and is close to zero if x represents zero. Essentially we think of $p(x)$ as giving the probability that the vector x represents an image of a one. If we want a definite choice, then we can set a threshold value p_0 and say that the image x is a one if $p(x) > p_0$ and zero otherwise. The natural choice of $p_0 = .5$ amounts to saying that we choose the more likely of the two options under the model.

Since we are applying the logistic model we are assuming:

- that the value of each pixel in the image contributes something towards the chance of the total image being one;
- and the different pixels have independent, and additive effects on the odds of getting a one.

If we take this point of view, then we can ask for the vector M that is *most likely* to account for the labellings, and we can use our maximum likelihood gradient descent method to find M.

This approach is surprisingly effective. With the MNIST zeros and ones, and the gradient descent method discussed above, one can easily find M so that the logistic model predicts the correct classification with accuracy in the high 90% range.

5.4.1 Weights as filters

One interesting aspect of using logistic regression on images for classification is that the we can interpret the optimum set of coefficients M as a kind of filter for our images. Remember that M is a vector with 785 entries, the last of which is an "intercept".

The logistic model says that, for an image vector x, the log-odds that the image is a one is given by

$$\log \frac{p}{1-p} = \sum_{i=0}^{783} M_i x_i + M_{784}.$$

This means that if the value of M_i is positive, then large values in the i^{th} pixel *increase* the chance that our image is a one; while if M_i is negative, large values *decrease* the chance. If M_i is negative, the reverse is true. However, the values x_i are the gray scale "darkness" of the image, so the entries of M emphasize or de-emphasize dark pixels according to whether that dark pixel is more or less likely to occur in a one compared to a zero.

This observation allows us to interpret the weights M as a kind of "filter" for the image. In fact, if we rescale the entries of M (omitting the intercept) so that they lie between 0 and 255, we can arrange them as a 28×28 array and plot them as an image. The result of doing this for a selection of MNIST zeros and ones is shown on the left in Figure 5.8. The red (or positive) weights in the middle of the image tell us that if those pixels are dark, the image is more likely to be a one; the blue (or negative) weights scattered farther out tell us that if those pixels are dark, the image is more likely to be a zero.

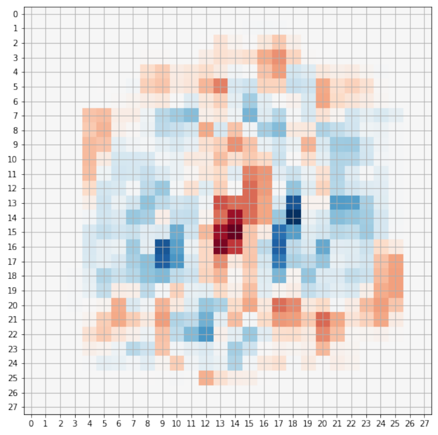
What's important to notice here is that we did not design this "filter" by hand, based on our understanding of the differences between a handwritten zero and one; instead, the algorithm "learned" the "best" filter to optimize its ability to distinguish these digits.

Here's another example. Suppose we redo the MNIST problem above, but we try to distinguish 3's from 8's.
We have about 4500 of each digit, and we label the 3's with zero and the 8's with one. Then we use our maximum likelihood optimization. In this case, the filter is shown on the bottom in Figure 5.8.

5.5 Multiclass Logistic Regression

One could attack the problem of classifying the ten distinct classes of digits by, for example, labelling all of the zeros as class zero and everything else as class one, and finding a set of weights that distinguishes zero from everything else. Then, in turn, one could do the same for each of the other digits. Given an unknown image, this would yield a set of probabilities from which one could choose the most likely class. This type of classification is called "one vs rest", for obvious reasons. It seems more natural, however, to construct a model that, given an image, assigns a probability that it belongs to each of the different possibilities. It is this type of multiclass logistic regression that we will study now.

Our goal is to build a model that, given an unknown image, returns a vector of ten probabilities, each of which we can interpret as the chance that our unknown image is in fact of a particular digit. If we *know* the image's class, then it's probability vector *should* be nine zeros with a



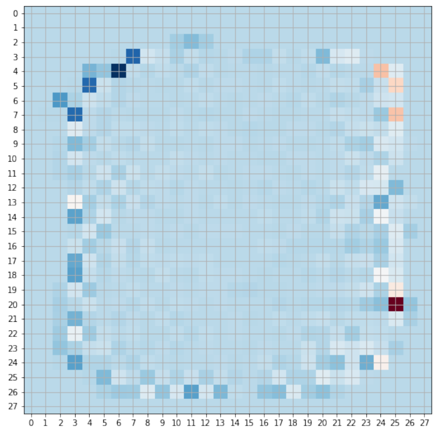


Figure 5.8: Rescaled weights (blue is negative). Top: 0 vs 1. Bottom: 3 vs 8.

single one in the position corresponding to the digit. So, for example, if our image is of a two, then the vector of probabilities

$$\begin{bmatrix} p_0 & p_1 & p_2 & \cdots & p_8 & p_9 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & \cdots & 0 & 0 \end{bmatrix}$$

where p_i is the probability that our image is the digit i. Notice also that the probabilities p_i must sum to one. We encode the class membership of our samples by constructing an $N \times r$ matrix Y, each row of which has a one in column j if that sample belongs to class j, and zeros elsewhere. This type of representation is sometimes called "one-hot" encoding.

So let's assume we have N data points, each with k features, and a one-hot encoded, $N \times r$ matrix of labels Y encoding the data into r classes. As usual, we add an "extra" feature, which is the constant 1 for each sample, to account for the "intercept". So our data matrix will be $N \times (k+1)$.

Our goal will be to find a $(k+1) \times r$ matrix of "weights" M so that, for each sample, we compute r values, given by the rows of the matrix XM. These r values are linear functions of the features, but we need probabilities. In the one-dimensional case, we used the logistic function σ to convert our linear function to probabilities. In this higher dimensional case we use a generalization of σ called the "softmax" function.

Definition: Let $F : \mathbf{R}^r \to \mathbf{R}^r$ be the function

$$F(z_1, \ldots, z_r) = \sum_{j=1}^{r} e^{z_i}$$

and let $\sigma : \mathbf{R}^r \to \mathbf{R}^r$ be the function

$$\sigma(z_1, \ldots, z_n) = \begin{bmatrix} \frac{e^{z_1}}{F} & \cdots & \frac{e^{z_r}}{F} \end{bmatrix}.$$

Notice that the coordinates of the vector $\sigma(z_1, \ldots, z_n)$ are all between 0 and 1, and their sum is one.

Our multiclass logistic model will say that the probability vector that gives the probabilities that a particular sample belongs to a particular class is given by the rows of the matrix $\sigma(XM)$, where $\sigma(XM)$ means applying the function σ to each row of the $N \times r$ matrix XM. For later computation, if:

- $x = X[i, :]$ is the $k+1$-entry feature vector of a single sample – a row of the data matrix X
- $m_j = M[:, j]$ is the $k+1$-entry column vector corresponding to the j^{th} column of M,

then the probability vector $[p_t]_{t=1}^{r}$ has entries

$$p_t(x; M) = \frac{e^{x \cdot m_t}}{\sum_{s=1}^{r} e^{x \cdot m_s}}.$$

5.5.1 Multiclass logistic regression - the likelihood

The probability vector $[p_t(x;M)]$ encodes the probabilities that the x-value belongs to each of the possible classes. That is,

$$p_j(x;M) = \text{The chance that x is in class j.}$$

We have captured the class membership of the samples in a $(k+1)\times r$ matrix Y which is "one-hot" encoded. Each row of this matrix has is zero in each place, except in the "correct" class, where it is one. Let $y = Y[i,:]$ be the i^{th} row of this matrix, so it is an r-entry row vector which is 1 in the position giving the "correct" class for our sample x.

So we can represent the chance that sample j belongs to class i as

$$P(\text{ sample i in class j}) = \prod_{s=1}^{r} p_s(x;M)^{y_s}.$$

Taking the logarithm, we find

$$\log P = \sum_{s=1}^{r} y_s \log p_s(x;M).$$

Since each sample is independent, the total likelihood is the product of these probabilites, and the log-likelihood the corresponding sum:

$$\log L(M) = \sum_{X,Y}\sum_{s=1}^{r} y_s \log p_s(x;M).$$

where the sum is over the N rows of X and Y. This is equivalent to the matrix expression

$$\log L(M) = \operatorname{trace}(Y^\top P) = \operatorname{trace}(Y^\top \sigma(XM))$$

This is the multiclass generalization of Equation 5.3. To see the connection, notice that, in the case where we have only two classes, $y_1 = 1 - y_0$ and $p_1(x;M) = 1 - p_0(x;M)$, so this sum is the same as in the two class situation.

5.5.2 Multiclass logistic regression - the gradient.

To find the "best-fitting" multiclass logistic model by gradient descent, we need an expression for the gradient of the likelihood $L(M)$. As with all of these calculations, this is an exercise in the chain rule. We start with the formula

$$p_s(x;M) = \frac{e^{x\cdot m_s}}{\sum_{t=1}^{r} e^{x\cdot m_t}}$$

The gradient of this is made up of the derivatives with respect to the m_{bq} where $b = 0, \ldots, k$ and $q = 1, \ldots, r$ so its natural to think of this gradient as a $(k+1) \times r$ matrix, the same shape as M. Remember that each m_s is the s^{th} column of M so is made up of m_{bs} for $b = 0, \ldots, k$.

Looking at

$$\frac{\partial p_s}{\partial m_{bq}}$$

there are two cases to consider. The first is when q and s are different, so the numerator of p_s doesn't involve m_{pq}. In this case the derivative is

$$\frac{\partial p_s}{\partial m_{bq}} = -\frac{e^{x \cdot m_s} e^{x \cdot m_q} x_b}{(\sum_{t=1}^{r} e^{x \cdot m_t})^2} = -p_s p_q x_b$$

In vector terms:

$$[\frac{\partial p_s}{\partial m_{bq}}]_{b=1}^{k} = -p_q p_s [x_b]_{b=1}^{k}$$

as an equality of $k+1$-entry row vectors. This can be written more simply as a vector equation:

$$\frac{\partial p_s}{\partial m_q} = -p_q p_s x. \qquad (q \neq s).$$

When $q = s$, we have

$$\frac{\partial p_s}{\partial m_{bs}} = \frac{e^{x \cdot m_{bs}} x_b}{\sum_{t=1}^{r} e^{x \cdot m_t}} - \frac{e^{x \cdot m_s} e^{x \cdot m_s} x_b}{(\sum_{t=1}^{r} e^{x \cdot m_t})^2} = p_s(1 - p_s) x_b$$

or in vector terms

$$\frac{\partial p_s}{\partial m_s} = p_s(1 - p_s) x^{\top}.$$

Important: The gradient on the left is properly seen as a column vector (because m_s is a column of the matrix M, with $k+1$ entries), and since x is a row of the data matrix, so to keep the indices straight, we need $x^{\top}$ on the right.

Now we can use these formulae together with the expression for $\log L(M)$ to obtain the gradient. Using the vector form, we have

$$\frac{\partial \log L(M)}{\partial m_q} = \sum_{X,Y} \sum_{s=1}^{r} y_s \frac{\partial \log p_s}{m_q}.$$

Using our computations above, the chain rule, and the derivative of the logarithm, this is the sum

$$\frac{\partial \log L(M)}{\partial m_q} = \sum_{X,Y} \sum_{s=1}^{r} y_s (I_{qs} - p_q) x^{\top}$$

where $I_{qs} = 1$ if $q = s$ and zero otherwise.

Now $y_s I_{qs}$ is zero unless $s = q$, and the sum $\sum_{s=1}^{r} y_s = 1$, so this simplifies further to

$$\frac{\partial \log L(M)}{\partial m_q} = \sum_{X,Y} (y_q - p_q) x^\top.$$

This is equivalent to the matrix expression

$$\nabla \log L(M) = X^\top (Y - P) = X^\top (Y - \sigma(XM)). \tag{5.4}$$

Compare Equation 5.4 to Theorem 5.1 and we see that the form is identical whether in the two-class or multi-class case if we set things up properly.

Algorithm: (Multiclass Gradient Descent) Given: - an $N \times (k+1)$ data matrix X whose last column is all 1, - an $N \times r$ matrix Y that "one-hot" encodes the labels of the classification problem; - a random $(k+1) \times r$ matrix M of initial guesses for the parameters - a "learning rate" ν,

Iterate:

$$M = M + \nu X^\top (Y - \sigma(XM))$$

until M changes by less than some tolerance.

5.6 Batch Descent

A look at the formulae for the gradient (see Equation 5.4) tells us that each iteratio

6 Support Vector Machines

6.1 Introduction

Suppose that we are given a collection of data made up of samples from two different classes, and we would like to develop an algorithm that can distinguish between the two classes. For example, given a picture that is either a dog or a cat, we'd like to be able to say which of the pictures are dogs, and which are cats. For another example, we might want to be able to distinguish "real" emails from "spam." This type of problem is called a *classification* problem.

Typically, one approaches a classification problem by beginning with a large set of data for which you know the classes, and you use that data to *train* an algorithm to correctly distinguish the classes for the test cases where you already know the answer. For example, you start with a few thousand pictures labelled "dog" and "cat" and you build your algorithm so that it does a good job distinguishing the dogs from the cats in this initial set of *training data.* Then you apply your algorithm to pictures that aren't labelled and rely on the predictions you get, hoping that whatever let your algorithm distinguish between the particular examples will generalize to allow it to correctly classify images that aren't pre-labelled.

Because classification is such a central problem, there are many approaches to it. We will see several of them through the course of these lectures. We will begin with a particular classification algorithm called "Support Vector Machines" (SVM) that is based on linear algebra. The SVM algorithm is widely used in practice and has a beautiful geometric interpretation, so it will serve as a good beginning for later discussion of more complicated classification algorithms.

Incidentally, I'm not sure why this algorithm is called a "machine"; the algorithm was introduced in the paper [6] where it is called the "Optimal Margin Classifier" and as we shall see that is a much better name for it.

My presentation of this material was heavily influenced by the beautiful paper [7].

6.2 A simple example

Let us begin our discussion with a very simple dataset (see [8] and [9]). This data consists of various measurements of physical characteristics of 344 penguins of 3 different species: Gentoo, Adelie, and Chinstrap. If we focus our attention for the moment on the Adelie and Gentoo species, and plot their body mass against their culmen depth, we obtain the following scatterplot.

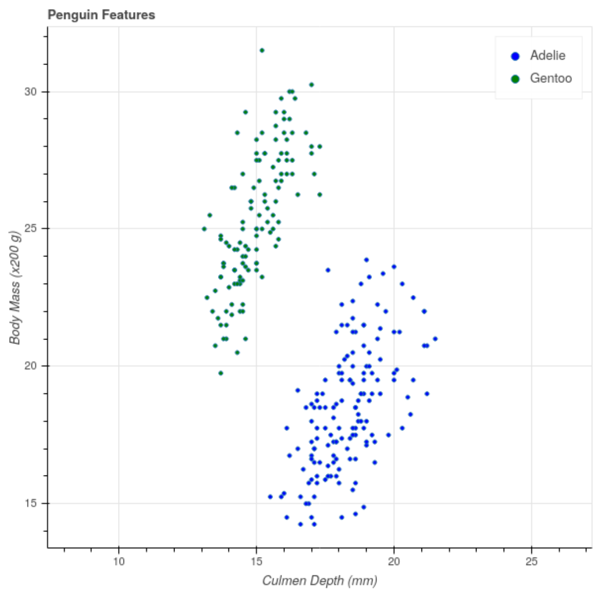


Figure 6.1: Penguin Scatterplot

Incidentally, a bird's *culmen* is the upper ridge of their beak, and the *culmen depth* is a measure of the thickness of the beak. There's a nice picture at [9] for the penguin enthusiasts.

A striking feature of this scatter plot is that there is a clear separation between the clusters of Adelie and Gentoo penguins. Adelie penguins have deeper culmens and less body mass than Gentoo penguins. These characteristics seem like they should provide a way to classify a penguin between these two species based on these two measurements.

One way to express the separation between these two clusters is to observe that one can draw a line on the graph with the property that all of the Adelie penguins lie on one side of that line and all of the Gentoo penguins lie on the other. In Figure 6.2 I've drawn in such a line (which I found by eyeballing the picture in Figure 6.1). The line has the equation

$$Y = 1.25X + 2.$$

The fact that all of the Gentoo penguins lie above this line means that, for the Gentoo penguins, their body mass in grams is at least 400 more than 250 times their culmen depth in mm. (Note that the y axis of the graph is scaled by 200 grams).

$$\text{Gentoo mass} > 250(\text{Gentoo culmen depth}) + 400$$

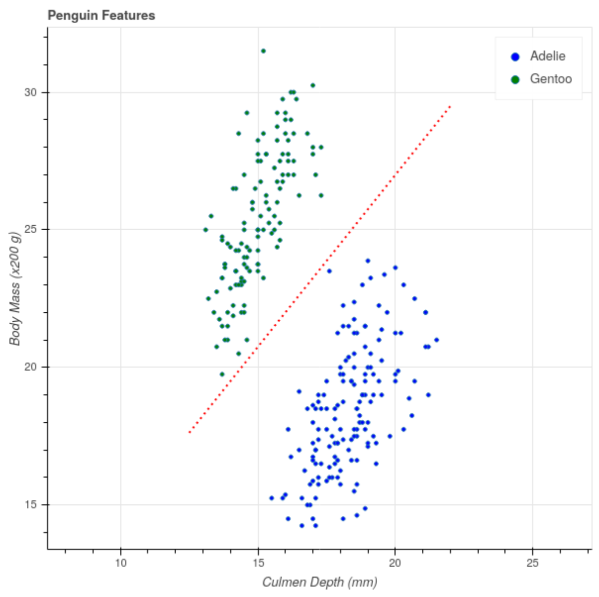


Figure 6.2: Penguins with Separating Line

while

$$\text{Adelie mass} < 250(\text{Adelie culmen depth}) + 400.$$

Now, if we measure a penguin caught in the wild, we can compute 250(culmen depth) + 400 for that penguin and if this number is greater than the penguin's mass, we say it's an Adelie; otherwise, a Gentoo. Based on the experimental data we've collected – the *training* data – this seems likely to work pretty well.

6.3 The general case

To generalize this approach, let's imagine now that we have n samples and k features (or measurements) for each sample. As before, we can represent this data as an $n \times k$ data matrix X. In the penguin example, our data matrix would be 344×2, with one row for each penguin and the columns representing the mass and the culmen depth. In addition to this numerical data, we have a classification that assigns each row to one of two classes. Let's represent the classes by a $n \times 1$ vector Y, where $y_i = +1$ if the i^{th} sample is in one class, and $y_i = -1$ if that i^{th} sample is in the other. Our goal is to predict Y based on X – but unlike in linear regression, Y takes on the values of ± 1.

In the penguin case, we were able to find a line that separated the two classes and then classify points by which side of the line the point was on. We can generalize this notion to higher dimensions. Before attacking that generalization, let's recall a few facts about the generalization to $\mathbf{R}^k$ of the idea of a line.

6.3.1 Hyperplanes

The correct generalization of a line given by an equation $w_1x_1 + w_2w_2 + b = 0$ in $\mathbf{R}^2$ is an equation $f(x) = 0$ where $f(x)$ is a degree one polynomial

$$f(x) = f(x_1, \dots, x_k) = w_1x_1 + w_2x_2 + \dots + w_kx_k + b \tag{6.1}$$

It's easier to understand the geometry of an equation like $f(x) = 0$ in Equation 6.1 if we think of the coefficients w_i as forming a *nonzero* vector $w = (w_1, \dots, w_k)$ in $\mathbf{R}^k$ and writing the formula for $f(x)$ as

$$f(x) = w \cdot x + b$$

.

Lemma: Let $f(x) = w \cdot x + b$ with $w \in \mathbf{R}^k$ a nonzero vector and b a constant in $\mathbf{R}$.

- The inequalities $f(x) > 0$ and $f(x) < 0$ divide up $\mathbf{R}^k$ into two disjoint subsets (called half spaces), in the way that a line in $\mathbf{R}^2$ divides the plane in half.
- The vector w is normal vector to the hyperplane $f(x) = 0$. Concretely this means that if p and q are any two points in that hyperplane, then $w \cdot (p - q) = 0$.
- Let $p = (u_1, \dots, u_k)$ be a point in $\mathbf{R}^k$. Then the perpendicular distance D from p to the hyperplane $f(x) = 0$ is
$$D = \frac{f(p)}{\|w\|}$$

Proof: The first part is clear since the inequalities are mutually exclusive. For the secon part, suppose that p and q satisfy $f(x) = 0$. Then $w \cdot p + b = w \cdot q + b = 0$. Subtracting these two equations gives $w \cdot (p - q) = 0$, so $p - q$ is orthogonal to w.

For the third part, consider Figure 6.3. The point q is an arbitrary point on the hyperplane defined by the equation $w \cdot x + b = 0$. The distance from the hyperplane to p is measured along the dotted line perpendicular to the hyperplane. The dot product $w \cdot (p - q) = \|w\|\|p - q\| \cos(\theta)$ where θ is the angle between $p - q$ and w – which is complementary to the angle between $p - q$ and the hyperplane. The distance D is therefore

$$D = \frac{w \cdot (p - q)}{\|w\|}.$$

However, since q lies on the hyperplane, we know that $w \cdot q + b = 0$ so $w \cdot q = -b$. Therefore $w \cdot (p - q) = w \cdot p + b = f(p)$, which is the formula we seek.

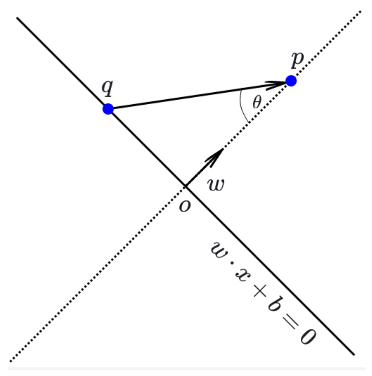


Figure 6.3: Distance to a Hyperplane

6.3.2 Linear separability and Margins

Now we can return to our classification scheme. The following definition generalizes our two dimensional picture from the penguin data.

Definition: Suppose that we have an $n \times k$ data matrix X and a set of labels Y that assign the n samples to one of two classes. Then the labelled data is said to be *linearly separable* if there is a vector w and a constant b so that, if $f(x) = w \cdot x + b$, then $f(x) > 0$ whenever $x = (x_1, \dots, x_k)$ is a row of X – a sample – belonging to the $+1$ class, and $f(x) < 0$ whenever x belongs to the -1 class. The solutions to the equation $f(x) = 0$ in this situation form a hyperplane that is called a *separating hyperplane* for the data.

In the situation where our data falls into two classes that are linearly separable, our classification strategy is to find a separating hyperplane f for our training data. Then, given a point x whose class we don't know, we can evaluate $f(x)$ and assign x to a class depending on whether $f(x) > 0$ or $f(x) < 0$.

This definition begs two questions about a particular dataset:

1. How do we tell if the two classes are linearly separable?
2. If the two sets are linearly separable, there are infinitely many separating hyperplanes. To see this, look back at the penguin example and notice that we can 'wiggle' the red line a little bit and it will still separate the two sets. Which is the 'best' separating hyperplane?

Let's try to make the first of these two questions concrete. We have two sets of points A and B in $\mathbf{R}^k$, and we want to (try to) find a vector w and a constant b so that $f(x) = w \cdot x + b$ takes strictly positive values for $x \in A$ and strictly negative ones for $x \in B$. Let's approach the problem by first choosing w and then asking whether there is a b that will work. In the two dimensional case, this is equivalent to choosing the slope of our line, and then asking if we can find an intercept so that the line passes between the two classes.

In algebraic terms, we are trying to solve the following system of inequalities: given w, find b so that:

$$w \cdot x + b > 0 \text{ for all } x \text{ in A}$$

and

$$w \cdot x + b < 0 \text{ for all } x \text{ in B.}$$

This is only going to be possible if there is a gap between the smallest value of $w \cdot x$ for $x \in A$ and the largest value of $w \cdot x$ for $x \in B$. In other words, given w there is a b so that $f(x) = w \cdot x + b$ separates A and B if

$$\max_{x \in B} w \cdot x < \min_{x \in A} w \cdot x.$$

If this holds, then choose b so that $-b$ lies in this open interval and you will obtain a separating hyperplane.

Proposition: The sets A and B are linearly separable if there is a w so that

$$\max_{x \in B} w \cdot x < \min_{x \in A} w \cdot x$$

If this inequality holds for some w, and $-b$ within this open interval, then $f(x) = w \cdot x + b$ is a separating hyperplane for A and B.

*Figure 6.4 is an illustration of this argument for a subset of the penguin data. Here, we have fixed $w = (1.25, -1)$ coming from the line $y = 1.25x + 2$ that we eyeballed earlier. For each Gentoo (green) point x_i, we computed $-b = w \cdot x_i$ and drew the line $f(x) = w \cdot x - w \cdot x_i$ giving a family of parallel lines through each of the green points. Similarly for each Adelie (blue) point we drew the corresponding line. The maximum value of $w \cdot x$ for the blue points turned out to be 1.998 and the minimum value of $w \cdot x$ for the green points turned out to be 2.003. Thus we have two lines with a gap between them, and any parallel line in that gap will separate the two sets.

Finally, among all the lines *with this particular* w, it seems that the **best** separating line is the one running right down the middle of the gap between the boundary lines. Any other line in the gap will be closer to either the blue or green set that the midpoint line is.

Let's put all of this together and see if we can make sense of it in general.

Suppose that A^+ and A^- are finite point sets in $\mathbf{R}^k$ and $w \in \mathbf{R}^k$ such that

$$B^-(w) = \max_{x \in A^-} w \cdot x < \min_{x \in A^+} w \cdot x = B^+(w).$$

Let x^- be a point in A^- with $w \cdot x^- = B^-(w)$ and x^+ be a point in A with $w \cdot x^+ = B^+(w)$. The two hyperplanes $f^\pm(x) = w \cdot x - B^\pm$ have the property that:

$$f^+(x) \geq 0 \text{ for } x \in A^+ \text{ and } f^+(x) < 0 \text{ for } x \in A^-$$

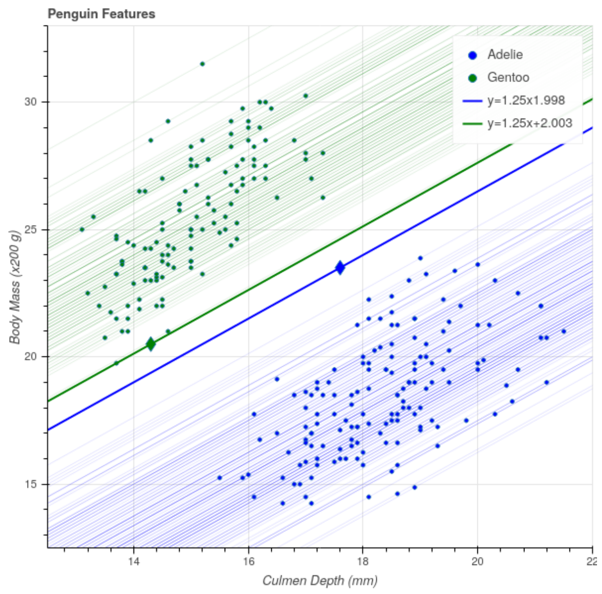


Figure 6.4: Lines in Penguin Data for $w = (1.25, -1)$

and

$$f^-(x) \leq 0 \text{ for } x \in A^- \text{ and } f^-(x) > 0 \text{ for } x \in A^+$$

Hyperplanes like f^+ and f^-, which "just touch" a set of points, are called supporting hyperplanes.

Definition: Let A be a set of points in $\mathbf{R}^k$. A hyperplane $f(x) = w \cdot x + b = 0$ is called a *supporting hyperplane* for A if $f(x) \geq 0$ for all $x \in A$ and $f(x) = 0$ for at least one point in A, or if $f(x) \leq 0$ for all $x \in A$ and $f(x) = 0$ for at least one point in A.

The gap between the two supporting hyperplanes f^+ and f^- is called the *margin* between A and B for w.

Definition: Let f^+ and f^- be as in the discussion above for point sets A^+ and A^- and vector w. Then the orthogonal distance between the two hyperplanes f^+ and f^- is called the geometric margin $\tau_w(A^+, A^-)$ (along w) between A^+ and A^-. We have

$$\tau_w(A^+, A^-) = \frac{B^+(w) - B^-(w)}{\|w\|}.$$

Now we can propose an answer to our second question about the best classifying hyperplane.

Definition: The *optimal margin* $\tau(A^+, A^-)$ between A^+ and A^- is the largest value of τ_w over all possible w for which $B^-(w) < B^+(w)$:

$$\tau(A^+, A^-) = \max_w \tau_w(A^+, A^-).$$

If w is such that $\tau_w = \tau$, then the hyperplane $f(x) = w \cdot x - \frac{(B^+ + B^-)}{2}$ is the *optimal margin classifying hyperplane.*

The optimal classifying hyperplane runs "down the middle" of the gap between the two supporting hyperplanes f^+ and f^- that give the sides of the optimal margin.

We can make one more observation about the maximal margin. If we find a vector w so that $f^+(x) = w \cdot x - B^+$ and $f^-(x) = w \cdot x - B^-$ are the two supporting hyperplanes such that the gap between them is the optimal margin, then this gap gives us an estimate on how close together the points in A^+ and A^- can be. This is visible in Figure 6.4, where it's clear that to get from a blue point to a green one, you have to cross the gap between the two supporting hyperplanes.

Proposition: The closest distance between points in A^+ and A^- is greater than or equal to the optimal margin:

$$\min_{p \in A^+, q \in A^-} \|p - q\| \geq \tau(A^+, A^-)$$

.

Proof: We have $f^+(p) = w \cdot p - B^+ \geq 0$ and $f^-(q) = w \cdot q - B^- \leq 0$. These two inequalities imply that

$$w \cdot (p - q) \geq B^+ - B^- > 0.$$

Therefore

$$\|p - q\| \|w\| \geq |w \cdot (p - q)| \geq |B^+ - B^-|$$

and so

$$\|p - q\| \geq \frac{B^+ - B^-}{\|w\|} = \tau(A^+, A^-)$$

If this inequality were always *strict* – that is, if the optimal margin equalled the minimum distance between points in the two clusters – then this would give us an approach to finding this optimal margin.

Unfortunately, that isn't the case. In Figure 6.5, we show a very simple case involving only six points in total in which the distance between the closest points in A^+ and A^- is larger than the optimal margin.

At least now our problem is clear. Given our two point sets A^+ and A^-, find w so that $\tau_w(A^+, A^-)$ is maximal among all w where $B^-(w) < B^+(w)$. This is an optimization problem, but unlike the optimization problems that arose in our discussions of linear regression and principal component analysis, it does not have a closed form solution. We will need to find an algorithm to determine w by successive approximations. Developing that algorithm will require thinking about a new concept known as *convexity.*

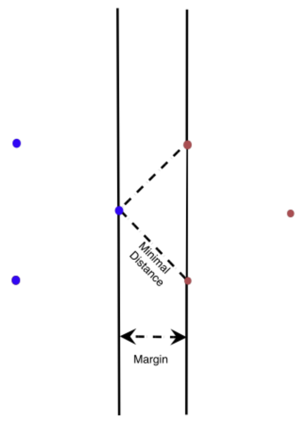


Figure 6.5: Shortest distance between + and - points can be greater than the optimal margin

6.4 Convexity, Convex Hulls, and Margins

In this section we introduce the notion of a *convex set* and the particular case of the *convex hull* of a finite set of points. As we will see, these ideas will give us a different interpretation of the margin between two sets and will eventually lead to an algorithm for finding the optimal margin classifier.

Definition: A subset U of $\mathbf{R}^k$ is *convex* if, for any pair of points p and q in U, every point t on the line segment joining p and q also belongs to U. In vector form, for every $0 \leq s \leq 1$, the point $t(s) = (1-s)p + sq$ belongs to U. (Note that $t(0) = p$, $t(1) = q$, and so $t(s)$ traces out the segment joining p to q.)

*Figure 6.6 illustrates the difference between convex sets and non-convex ones.

The key idea from convexity that we will need to solve our optimization problem and find the optimal margin is the idea of the *convex hull* of a finite set of points in $\mathbf{R}^k$.

Definition: Let $S = \{q_1, \dots, q_N\}$ be a finite set of N points in $\mathbf{R}^k$. The *convex hull* $C(S)$ of S is the set of points

$$p = \sum_{i=1}^{N} \lambda_i q_i$$

as $\lambda_1, \dots, \lambda_N$ runs over all positive real numbers such that

$$\sum_{i=1}^{N} \lambda_i = 1.$$

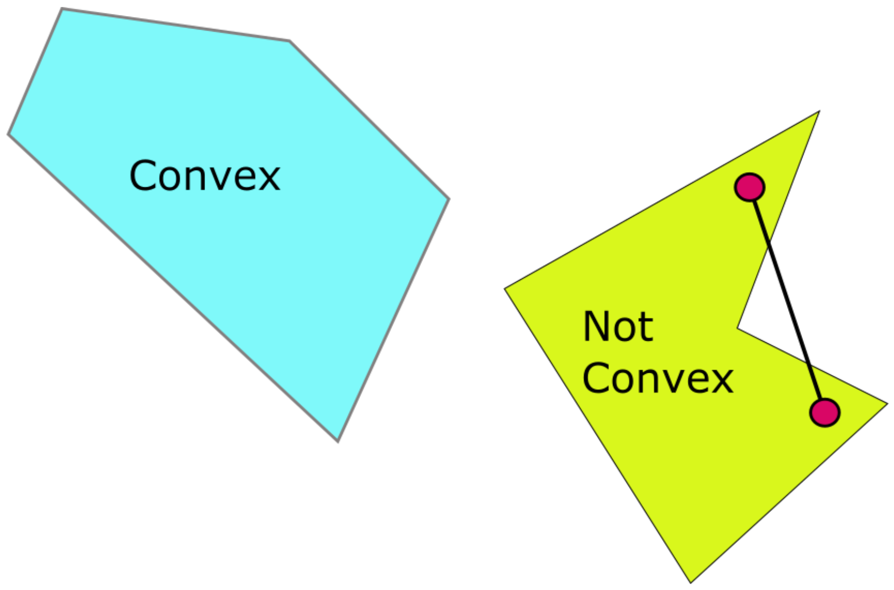


Figure 6.6: Convex vs Non-Convex Sets

There are a variety of ways to think about the convex hull $C(S)$ of a set of points S, but perhaps the most useful is that it is the smallest convex set that contains all of the points of S. That is the content of the next lemma.

Lemma: $C(S)$ is convex. Furthermore, let U be any convex set containing all of the points of S. Then U contains $C(S)$.

Proof: To show that $C(S)$ is convex, we apply the definition. Let p_1 and p_2 be two points in $C(S)$, so that let $p_j = \sum_{i=1}^{N} \lambda_i^{(j)} q_i$ where $\sum_{i=1}^{N} \lambda_i^{(j)} = 1$ for $j = 1, 2$. Then a little algebra shows that

$$(1-s)p_1 + sp_2 = \sum_{i=1}^{N} (s\lambda_i^{(1)} + (1-s)\lambda_i^{(2)}) q_i$$

and $\sum_{i=1}^{N} (s\lambda_i^{(1)} + (1-s)\lambda_i^{(2)}) = 1$. Therefore all of the points $(1-s)p_1 + sp_2$ belong to $C(S)$, and therefore $C(S)$ is convex.

For the second part, we proceed by induction. Let U be a convex set containing S. Then by the definition of convexity, U contains all sums $\lambda_i q_i + \lambda_j q_j$ where $\lambda_i + \lambda_j = 1$. Now suppose that U contains all the sums $\sum_{i=1}^{N} \lambda_i q_i$ where exactly $m-1$ of the λ_i are non-zero for some $m < N$.

Consider a sum

$$q = \sum_{i=1}^{N} \lambda_i q_i$$

with exactly m of the $\lambda_i \neq 0$. For simplicity let's assume that $\lambda_i \neq 0$ for $i = 1, \dots, m$. Now let $T = \sum_{i=1}^{m-1} \lambda_i$ and set

$$q' = \sum_{i=1}^{m-1} \frac{\lambda_i}{T} q_i.$$

This point q' belongs to U by the inductive hypothesis. Also, $(1 - T) = \lambda_m$. Therefore by convexity of U,

$$q = (1 - T)q_m + Tq'$$

also belongs to U. It follows that all of $C(S)$ belongs to U.

In Figure 6.7 we show our penguin data together with the convex hull of points corresponding to the two types of penguins. Notice that the boundary of each convex hull is a finite collection of line segments that join the "outermost" points in the point set.

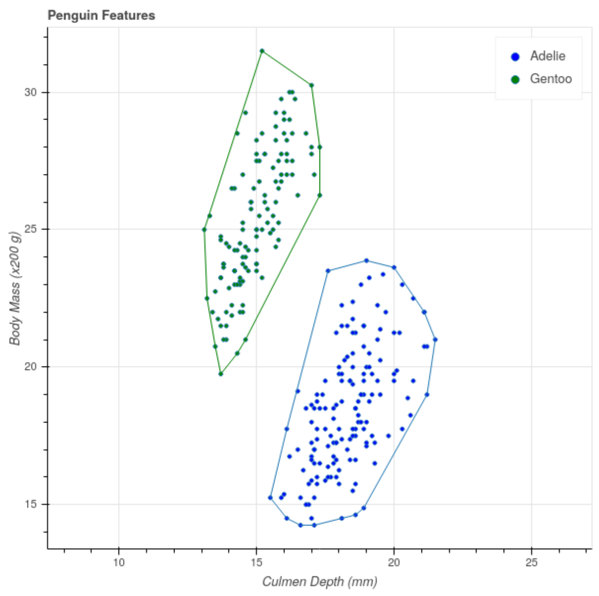


Figure 6.7: The Convex Hull

One very simple example of a convex set is a half-plane. More specifically, if $f(x) = w \cdot x + b = 0$ is a hyperplane, then the two "sides" of the hyperplane, meaning the subsets $\{x : f(x) \geq 0\}$ and $\{x : f(x) \leq 0\}$, are both convex. (This is exercise 1 in Section 6.7).

As a result of this observation, and the Lemma above, we can conclude that if $f(x) = w \cdot x + b = 0$ is a supporting hyperplane for the set S – meaning that either $f(x) \geq 0$ for all $x \in S$, or $f(x) \leq 0$ for all $x \in S$, with at least one point $x \in S$ such that $f(x) = 0$ – then $f(x) = 0$ is a supporting hyperplane for the entire convex hull. After all, if $f(x) \geq 0$ for all points $x \in S$, then S is contained in the convex set of points where $f(x) \geq 0$, and therefore $C(S)$ is contained in that set as well.

Interestingly, however, the converse is true as well – the supporting hyperplanes of $C(S)$ are exactly the same as those for S.

Lemma: Let S be a finite set of points in $\mathbf{R}^k$ and let $f(x) = w \cdot x + b = 0$ be a supporting hyperplane for $C(S)$. Then $f(x)$ is a supporting hyperplane for S.

Proof: Suppose $f(x) = 0$ is a supporting hyperplane for $C(S)$. Let's assume that $f(x) \geq 0$ for all $x \in C(S)$ and $f(x^*) = 0$ for a point $x^* \in C(S)$, since the case where $f(x) \leq 0$ is identical. Since $S \subset C(S)$, we have $f(x) \geq 0$ for all $x \in S$. To show that $f(x) = 0$ is a supporting hyperplane, we need to know that $f(x) = 0$ for at least one point $x \in S$.
Let x' be the point in S where $f(x')$ is minimal among all $x \in S$. Note that $f(x') \geq 0$. Then the hyperplane $g(x) = f(x) - f(x')$ has the property that $g(x) \geq 0$ on all of S, and $g(x') = 0$. Since the halfplane $g(x) \geq 0$ is convex and contains all of S, we have $C(S)$ contained in that halfplane. So, on the one hand we have $g(x^*) = f(x^*) - f(x') \geq 0$. On the other hand $f(x^*) = 0$, so $-f(x') \geq 0$, so $f(x') \leq 0$. Since $f(x')$ is also greater or equal to zero, we have $f(x') = 0$, and so we have found a point of S on the hyperplane $f(x) = 0$. Therefore $f(x) = 0$ is also a supporting hyperplane for S.

This argument can be used to give an alternative description of $C(S)$ as the intersection of all halfplanes containing S arising from supporting hyperplanes for S. This is exercise 2 in Section 6.7. It also has as a corollary that $C(S)$ is a closed set.

Lemma: $C(S)$ is compact.

Proof: Exercise 2 in Section 6.7 shows that it is the intersection of closed sets in $\mathbf{R}^k$, so it is closed. Exercise 3 shows that $C(S)$ is bounded. Thus it is compact.

Now let's go back to our optimal margin problem, so that we have linearly separable sets of points A^+ and A^-. Recall that we showed that the optimal margin was at most the minimal distance between points in A^+ and A^-, but that there could be a gap between the minimal distance and the optimal margin – see Figure 6.5 for a reminder.

It turns out that by considering the minimal distance between $C(A^+)$ and $C(A^-)$, we can "close this gap." The following proposition shows that we can change the problem of finding the optimal margin into the problem of finding the closest distance between the convex hulls of $C(A^+)$ and $C(A^-)$. The following proposition generalizes the Proposition at the end of Section 6.3.2.

Proposition: Let A^+ and A^- be linearly separable sets in $\mathbf{R}^k$. Let $p \in C(A^+)$ and $q \in C(A^-)$ be any two points. Then

$$\|p - q\| \geq \tau(A^+, A^-).$$

Proof: As in the earlier proof, choose supporting hyperplanes $f^+(x) = w \cdot x - B^+ = 0$ and $f^-(x) = w \cdot x - B^-$ for A^+ and A^-. By our discussion above, these are also supporting hyperplanes for $C(A^+)$ and $C(A^-)$. Therefore if $p \in C(A^+)$ and $q \in C(A^-)$, we have $w \cdot p - B^+ \geq 0$ and $w \cdot q - B^- \leq 0$. As before

$$w \cdot (p - q) \geq B^+ - B^- > 0$$

and so

$$\|p - q\| \geq \frac{B^+ - B^-}{\|w\|} = \tau_w(A^+, A^-)$$

Since this holds for any w, we have the result for $\tau(A^+, A^-)$.

The reason this result is useful is that, as we've seen, if we restrict p and q to A^+ and A^-, then there can be a gap between the minimal distance and the optimal margin. If we allow p and q to range over the convex hulls of these sets, then that gap disappears.

One other consequence of this is that if A^+ and A^- are linearly separable then their convex hulls are disjoint.

Corollary: If A^+ and A^- are linearly separable then $\|p - q\| > 0$ for all $p \in C(A^+)$ and $q \in C(A^-)$

Proof: The sets are linearly separable precisely when $\tau > 0$.

Our strategy now is to show that if p and q are points in $C(A^+)$ and $C(A^-)$ respectively that are at minimal distance D, and if we set $w = p - q$, then we obtain supporting hyperplanes with margin equal to $\|p - q\|$. Since this margin is the *largest possible margin*, this w must be the optimal w. This transforms the problem of finding the optimal margin into the problem of finding the closest points in the convex hulls.

Lemma: Let

$$D = \min_{p \in C(A^+), q \in C(A^-)} \|p - q\|.$$

Then there are points $p^* \in C(A^+)$ and $q^* \in C(A^-)$ with $\|p^* - q^*\| = D$. If p_1^*, q_1^* and p_2^*, q_2^* are two pairs of points satisfying this condition, then $p_1^* - q_1^* = p_2^* - q_2^*$.

Proof: Consider the set of differences

$$V = \{p - q : p \in C(A^+), q \in C(A^-)\}.$$

- V is compact. This is because it is the image of the compact set $C(A^+) \times C(A^-)$ in $\mathbf{R}^k \times \mathbf{R}^k$ under the continuous map $h(x, y) = x - y$.
- the function $d(v) = \|v\|$ is continuous and satisfies $d(v) \geq D > 0$ for all $v \in V$.

Since d is a continuous function on a compact set, it attains its minimum D and so there is a $v = p^* - q^*$ with $d(v) = D$.

Now suppose that there are two distinct points $v_1 = p_1^* - q_1^*$ and $v_2 = p_2^* - q_2^*$ with $d(v_1) = d(v_2) = D$. Consider the line segment

$$t(s) = (1-s)v_1 + sv_2 \text{ where } 0 \leq s \leq 1$$

joining v_1 and v_2.
Now

$$t(s) = ((1-s)p_1^* + sp_2^*) - ((1-s)q_1^* + sq_2^*).$$

Both terms in this difference belong to $C(A^+)$ and $C(A^-)$ respectively, regardless of s, by convexity, and therefore $t(s)$ belongs to V for all $0 \leq s \leq 1$.

This little argument shows that V is convex. In geometric terms, v_1 and v_2 are two points in the set V equidistant from the origin and the segment joining them is a chord of a circle; as Figure 6.8 shows, in that situation there must be a point on the line segment joining them that's closer to the origin than they are. Since all the points on that segment are in V by convexity, this would contradict the assumption that v_1 is the closet point in V to the origin.

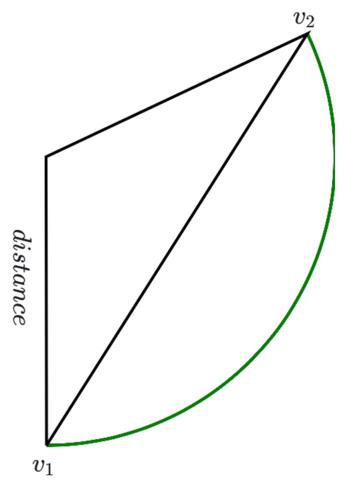


Figure 6.8: Chord of a circle

In algebraic terms, since D is the minimal value of $\|v\|$ for all $v \in V$, we must have $t(s) \geq D$. On the other hand

$$\frac{d}{ds}\|t(s)\|^2 = \frac{d}{ds}(t(s) \cdot t(s)) = t(s) \cdot \frac{dt(s)}{ds} = t(s) \cdot (v_2 - v_1).$$

Therefore

$$\frac{d}{ds}\|t(s)\|^2|_{s=0} = v_1 \cdot (v_2 - v_1) = v_1 \cdot v_2 - \|v_1\|^2 \leq 0$$

since $v_1 \cdot v_2 \leq D^2$ and $\|v_1\|^2 = D^2$. If $v_1 \cdot v_2 < D^2$, then this derivative would be negative, which would mean that there is a value of s where $t(s)$ would be less than D. Since that can't happen, we conclude that $v_1 \cdot v_2 = D^2$ which means that $v_1 = v_2$ – the vectors have the same magnitude D and are parallel. This establishes uniqueness.

Note: The essential ideas of this argument show that a compact convex set in $\mathbf{R}^k$ has a unique point closest to the origin. The convex set in this instance,

$$V = \{p - q : p \in C(A^+), q \in C(A^-)\},$$

is called the difference $C(A^+) - C(A^-)$, and it is generally true that the difference of convex sets is convex.

Now we can conclude this line of argument.

Theorem: Let p and q be points in $C(A^+)$ and $C(A^-)$ respectively are such that $\|p - q\|$ is minimal among all such pairs. Let $w = p - q$ and set $B^+ = w \cdot p$ and $B^- = w \cdot q$. Then $f^+(x) = w \cdot x - B^+ = 0$ and $f^-(x) = w \cdot x - B^-$ are supporting hyperplanes for $C(A^+)$ and $C(A^-)$ respectively and the associated margin

$$\tau_w(A^+, A^-) = \frac{B^+ - B^-}{\|w\|} = \|p - q\|$$

is optimal.

Proof: First we show that $f^+(x) = 0$ is a supporting hyperplane for $C(A^+)$. Suppose not. Then there is a point $p' \in C(A^+)$ such that $f^+(x) < 0$. Consider the line segment $t(s) = (1-s)p + sp'$ running from p to p'. By convexity it is entirely contained in $C(A^+)$. Now look at the distance from points on this segment to q:

$$D(s) = \|t(s) - q\|^2.$$

We have

$$\frac{dD(s)}{ds}|_{s=0} = 2(p - q) \cdot (p' - p) = 2w \cdot (p' - p) = 2\left[(f^+(p') + B^+) - (f^+(p) + B^+)\right]$$

so

$$\frac{dD(s)}{ds}|_{s=0} = 2(f^+(p') - f^+(p)) < 0$$

since $f(p) = 0$. This means that $D(s)$ is decreasing along $t(s)$ and so there is a point s' along $t(s)$ where $\|t(s') - q\| < D$. This contradicts the fact that D is the minimal distance. The same argument shows that $f^-(x) = 0$ is also a supporting hyperplane.

Now the margin for this w is

$$\tau_w(A^+, A^-) = \frac{w \cdot (p - q)}{\|w\|} = \|p - q\| = D$$

and as w varies we know this is the largest possible τ that can occur. Thus this is the maximal margin.

*Figure 6.9 shows how considering the closest point in the convex hulls "fixes" the problem that we saw in Figure 6.5. The closest point occurs at a point on the boundary of the convex hull that is not one of the points in A^+ or A^-.

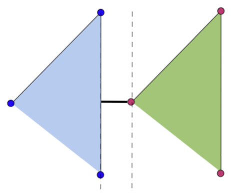


Figure 6.9: Closest distance between convex hulls gives optimal margin

6.5 Finding the Optimal Margin Classifier

Now that we have translated our problem into geometry, we can attempt to develop an algorithm for solving it. To recap, we have two sets of points

$$A^+ = \{x_1^+, \dots, x_{n_+}^+\}$$

and

$$A^- = \{x_1^-, \dots, x_{n_-}^-\}$$

in $\mathbf{R}^k$ that are linearly separable.

We wish to find points $p \in C(A^+)$ and $q \in C(A^-)$ such that

$$\|p - q\| = \min_{p' \in C(A^+), q' \in C(A^-)} \|p' - q'\|.$$

Using the definition of the convex hull we can express this more concretely. Since $p \in C(A^+)$, there are coefficients $\lambda_i^+ \geq 0$ for $i = 1, \dots, n_+$ and $\lambda_i^- \geq 0$ for $i = 1, \dots, n_-$ so that

$$p = \sum_{i=1}^{n_+} \lambda_i^+ x_i^+$$

$$q = \sum_{i=1}^{n_-} \lambda_i^- x_i^-$$

where $\sum_{i=1}^{n_\pm} \lambda_i^\pm = 1$.

We can summarize this as follows:

Optimization Problem 1: Write $\lambda^\pm = (\lambda_1^\pm, \dots, \lambda_{n_\pm}^\pm)$ Define

$$w(\lambda^+, \lambda^-) = \sum_{i=1}^{n_+} \lambda_i^+ x_i^+ - \sum_{i=1}^{n_-} \lambda^- x_i^-$$

To find the supporting hyperplanes that define the optimal margin between A^+ and A^-, find λ^+ and λ^- such that $\|w(\lambda^+, \lambda^-)\|^2$ is minimal among all such w where all $\lambda_i^\pm \geq 0$ and $\sum_{i=1}^{n_\pm} \lambda_i^\pm = 1$.

This is an example of a *constrained optimization problem.* It's worth observing that the *objective function* $\|w(\lambda^+, \lambda^-)\|^2$ is just a quadratic function in the $\lambda^\pm$. Indeed we can expand

$$\|w(\lambda^+, \lambda^-)\|^2 = (\sum_{i=1}^{n_+} \lambda_i^+ x_i - \sum_{i=1}^{n_-} \lambda^- x_i^-) \cdot (\sum_{i=1}^{n_+} \lambda_i^+ x_i - \sum_{i=1}^{n_-} \lambda^- x_i^-)$$

to obtain

$$\|w(\lambda^+, \lambda^-)\|^2 = R - 2S + T$$

where

$$\begin{aligned} R =& \sum_{i=1}^{n_+} \sum_{j=1}^{n_+} \lambda_i^+ \lambda_j^+ (x_i^+ \cdot x_j^+) \\ S =& \sum_{i=1}^{n_+} \sum_{j=1}^{n_-} \lambda_i^+ \lambda_j^- (x_i^+ \cdot x_j^-) \\ T =& \sum_{i=1}^{n_-} \sum_{j=1}^{n_-} \lambda_i^- \lambda_j^- (x_i^- \cdot x_j^-) \end{aligned} \tag{6.2}$$

Thus the function we are trying to minimize is relatively simple.

On the other hand, unlike optimization problems we have seen earlier in these lectures, in which we can apply Lagrange multipliers, in this case some of the constraints are inequalities – namely the requirement that all of the $\lambda^\pm \geq 0$ – rather than equalities. There is an extensive theory of such problems that derives from the idea of Lagrange multipliers. However, in these notes, we will not dive into that theory but will instead construct an algorithm for solving the problem directly.

6.5.1 Relaxing the constraints

Our first step in attacking this problem is to adjust our constraints and our objective function slightly so that the problem becomes easier to attack.

Optimization Problem 2: This is a slight revision of problem 1 above. We minimize:

$$Q(\lambda^+, \lambda^-) = \|w(\lambda^+, \lambda^-)\|^2 - \sum_{i=1}^{n_+} \lambda_i^+ - \sum_{i=1}^{n_-} \lambda_i^-$$

subject to the constraints that all $\lambda_i^\pm \geq 0$ and

$$\alpha = \sum_{i=1}^{n_+} \lambda_i^+ = \sum_{i=1}^{n_-} \lambda_i^-.$$

Problem 2 is like problem 1, except we don't require the sums of the $\lambda_i^\pm$ to be one, but only that they be equal to each other; and we modify the objective function slightly. It turns out that the solution to this optimization problem easily yields the solution to our original one.

Lemma: Suppose λ^+ and λ^- satisfy the constraints of problem 2 and yield the minimal value for the objective function $Q(\lambda^+, \lambda^-)$. Then $\alpha \neq 0$. Rescale the $\lambda^\pm$ to have sum equal to one by dividing by α, yielding $\tau^\pm = (1/\alpha)\lambda^\pm$. Then $w(\tau^+, \tau^-)$ is a solution to optimization problem 1.

Proof: To show that $\alpha \neq 0$, suppose that $\lambda_i^\pm = 0$ for all $i \neq 1$ and $\lambda = \lambda_1^+ = \lambda_1^-$. The one-variable quadratic function $Q(\lambda)$ takes its minimum value at $\lambda = 1/\|x_1^+ - x_1^-\|^2$ and its value at that point is negative. Therefore the minimum value of Q is negative, which means $\alpha \neq 0$ at that minimum point.

For the equivalence, notice that $\tau^\pm$ still satisfy the constraints of problem 2. Therefore

$$Q(\lambda^+, \lambda^-) = \|w(\lambda^+, \lambda^-)\|^2 - 2\alpha \leq \|w(\tau^+, \tau^-)\|^2 - 2.$$

On the other hand, suppose that $\sigma^\pm$ are a solution to problem 1. Then

$$\|w(\sigma^+, \sigma^-)\|^2 \leq \|w(\tau^+, \tau^-)\|^2.$$

Therefore

$$\alpha^2\|w(\sigma^+, \sigma^-)\|^2 = \|w(\alpha\sigma^+, \alpha\sigma^-)\|^2 \leq \|w(\lambda^+, \lambda^-)\|^2$$

and finally

$$\|w(\alpha\sigma^+, \alpha\sigma^-)\|^2 - 2\alpha \leq Q(\lambda^+, \lambda^-) = \|w(\alpha\tau^+, \alpha\tau^-)\|^2 - 2\alpha.$$

Since Q is the minimal value, we have

$$\alpha^2\|w(\sigma^+, \sigma^-)\|^2 = \alpha^2\|w(\tau^+, \tau^-)\|^2$$

so that indeed $w(\tau^+, \tau^-)$ gives a solution to Problem 1.

6.5.2 Sequential Minimal Optimization

Now we outline an algorithm for solving Problem 2 that is called Sequential Minimal Optimization that was introduced by John Platt in 1998 (See [10] and Chapter 12 of [11]). The algorithm is based on the principle of "gradient ascent", where we exploit the fact that the negative gradient of a function points in the direction of its most rapid decrease and we take small steps in the direction of the negative gradient until we reach the minimum.

However, in this case simplify this idea a little. Recall that the objective function $Q(\lambda^+, \lambda^-)$ is a quadratic function in the λ's and that we need to preserve the condition that $\sum \lambda_i^+ = \sum \lambda_i^-$. So our approach is going to be to take, one at a time, a pair λ_i^+ and λ_j^- and change them *together* so that the equality of the sums is preserved and the change reduces the value of the objective function. Iterating this will take us to a minimum.

So, for example, let's look at λ_i^+ and λ_j^- and, for the moment, think of all of the other λ's as constants. Then our objective function reduces to a quadratic function of these two variables that looks something like:

$$Q(\lambda_i^+, \lambda_j^-) = a(\lambda_i^+)^2 + b\lambda_i^+\lambda_j^- + c(\lambda_i^-)^2 + d\lambda_i^+ + e\lambda_j^- + f.$$

The constraints that remain are $\lambda^\pm \geq 0$, and we are going to try to minimize Q by changing λ_i^+ and λ_j^- *by the same amount* δ. Furthermore, since we still must have $\lambda_i^+ + \delta \geq 0$ and $\lambda_j^- + \delta \geq 0$, we have

$$\delta \geq M = \max\{-\lambda_i^+, -\lambda_j^-\} \tag{6.3}$$

In terms of this single variable δ, our optimization problem becomes the job of finding the minimum of a quadratic polynomial in one variable subject to the constraint in Equation 6.3. This is easy! There are two cases: the critical point of the quadratic is to the left of M, in which case the minimum value occurs at M; or the critical point of the quadratic is to the right of M, in which case the critical point occurs there. This is illustrated in Figure 6.10.

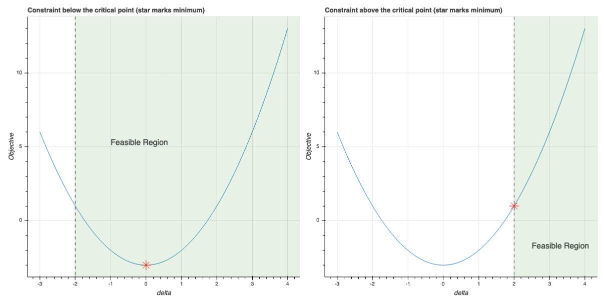


Figure 6.10: Minimizing the 1-variable quadratic objective function

Computationally, let's write

$$w_{\delta,i,j}(\lambda^+,\lambda^-) = w(\lambda^+,\lambda^-) + \delta(x_i^+ - x_j^-).$$

Then

$$\frac{d}{d\delta}(\|w_{\delta,i,j}(\lambda^+,\lambda^-)\|^2 - 2\alpha) = 2w_{\delta,i,j}(\lambda^+,\lambda^-)\cdot(x_i^+ - x_j^-) - 2$$

and using the definition of $w_{\delta,i,j}$ we obtain the following formula for the critical value of δ by setting this derivative to zero:

$$\delta_{i,j} = \frac{(1 - w(\lambda^+,\lambda^-)\cdot(x_i^+ - x_j^-)}{\|x_i^+ - x_j^-\|^2}$$

Using this information we can describe the SMO algorithm.

Algorithm (SMO, see [10]):

Given: Two linearly separable sets of points $A^+ = \{x_1^+,\dots,x_{n_+}^+\}$ and $A^- = \{x_1^-,\dots,x_{n_-}^-\}$ in $\mathbf{R}^k$.

Find: Points p and q belonging to $C(A^+)$ and $C(A^-)$ respectively such that

$$\|p-q\|^2 = \min_{p'\in C(A^+),q'\in C(A^-)} \|p'-q'\|^2$$

Initialization: Set $\lambda_i^+ = \frac{1}{n_+}$ for $i = 1,\dots,n_+$ and $\lambda_i^- = \frac{1}{n_-}$ for $i = 1,\dots,n_-$. Set

$$p(\lambda^+) = \sum_{i=1}^{n_+} \lambda_i^+ x_i^+$$

and

$$q(\lambda^-) = \sum_{i=1}^{n_-} \lambda_i^- x_i^-$$

Notice that $w(\lambda^+,\lambda^-) = p(\lambda^+) - q(\lambda^-)$. Let $\alpha = \sum_{i=1}^{n_+}\lambda^+ = \sum_{i=1}^{n_-}\lambda^-$. These sums will remain equal to each other throughout the operation of the algorithm.

Repeat the following steps until maximum value of δ^* computed in each iteration is smaller than some tolerance (so that the change in all of the λ's is very small):

- For each pair i,j with $1 \le i \le n_+$ and $1 \le j \le n_-$, compute

  $$M_{i,j} = \max\{-\lambda_i^+, -\lambda_j^-\}$$

  and

  $$\delta_{i,j} = \frac{1 - (p(\lambda^+) - q(\lambda^-))\cdot(x_i^+ - x_j^-)}{\|x_i^+ - x_j^-\|^2}.$$

If $\delta_{i,j} \geq M$ then set $\delta^* = \delta_{i,j}$; otherwise set $\delta^* = M$. Then update the $\lambda^{\pm}$ by the equations:

$$\begin{aligned}\lambda_i^+ &= \lambda_i^+ + \delta_{i,j}^* \\ \lambda_j^+ &= \lambda_j^- + \delta_{i,j}^*\end{aligned}$$

When this algorithm finishes, $p \approx p(\lambda^+)$ and $q \approx q(\lambda^-)$ will be very good approximations to the desired closest points.

Recall that if we set $w = p - q$, then the optimal margin classifier is

$$f(x) = w \cdot x - \frac{B^+ + B^-}{2} = 0$$

where $B^+ = w \cdot p$ and $B^- = w \cdot q$. Since $w = p - q$ we can simplify this to obtain

$$f(x) = (p - q) \cdot x - \frac{\|p\|^2 - \|q\|^2}{2} = 0.$$

In Figure 6.11, we show the result of applying this algorithm to the penguin data and illustrate the closest points as found by an implementation of the SMO algorithm, together with the optimal classifying line.

Bearing in mind that the y-axis is scaled by a factor of 200, we obtain the following rule for distinguishing between Adelie and Gentoo penguins – if the culmen depth and body mass put you above the red line, you are a Gentoo penguin, otherwise you are an Adelie.

6.6 Inseparable Sets

Not surprisingly, real life is often more complicated than the penguin example we've discussed at length in these notes. In particular, sometimes we have to work with sets that are not linearly separable. Instead, we might have two point clouds, the bulk of which are separable, but because of some outliers there is no hyperplane we can draw that separates the two sets into two halfplanes.

Fortunately, all is not lost. There are two common ways to address this problem, and while we won't take the time to develop the theory behind them, we can at least outline how they work.

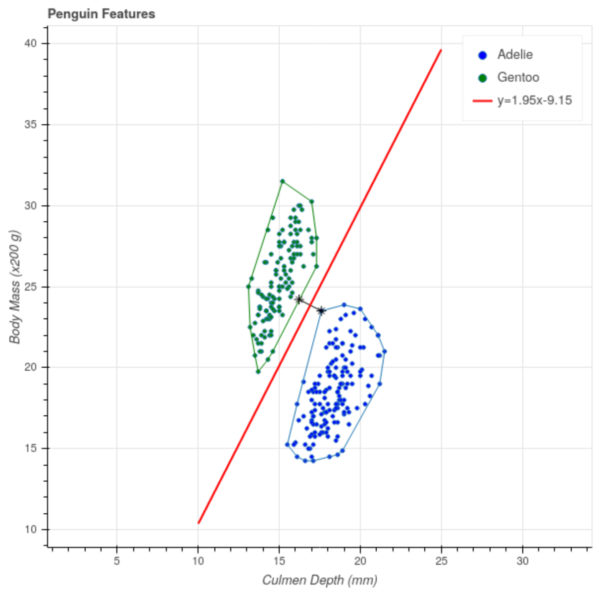


Figure 6.11: Closest points in convex hulls of penguin data

6.6.1 Best Separating Hyperplanes

If our sets are not linearly separable, then their convex hulls overlap and so our technique for finding the closest points of the convex hulls won't work. In this case, we can "shrink" the convex hull by considering combinations of points $\sum_i \lambda_i x_i$ where $\sum \lambda_i = 1$ and $C \geq \lambda_i \geq 0$ for some $C \leq 1$. For C small enough, reduced convex hulls will be linearly separable – although some outlier points from each class will lie outside of them – and we can find hyperplane that separates the reduced hulls.
In practice, this means we allow a few points to lie on the "wrong side" of the hyperplane. Our tolerance for these mistakes depends on C, but we can include C in the optimization problem to try to find the smallest C that "works".

6.6.2 Nonlinear kernels

The second option is to look not for separating hyperplanes but instead for separating curves – perhaps polynomials or even more exotic curves. This can be achieved by taking advantage of the form of Equation 6.2. As you see there, the only way the points $x_i^{\pm}$ enter in to the function being minimized is through the inner products $x_i^{\pm} \cdot x_j^{\pm}$. We can adopt a different inner product than the usual Euclidean one, and reconsider the problem using this different inner product. This amounts to embedding our points in a higher dimensional space where they are more likely to be linearly separable. Again, we will not pursue the mathematics of this further in these notes.

6.7 Exercises

1. Prove that, if $f(x) = w \cdot x + b = 0$ is a hyperplane in $\mathbf{R}^k$, then the two "sides" of this hyperplane, consisting of the points where $f(x) \geq 0$ and $f(x) \leq 0$, are both convex sets.
2. Prove that $C(S)$ is the intersection of all the halfplanes $f(x) \geq 0$ as $f(x) = w \cdot x + b$ runs through all supporting hyperplanes for S where $f(x) \geq 0$ for all $p \in S$.
3. Prove that $C(S)$ is bounded. Hint: show that S is contained in a sphere of sufficiently large radius centered at zero, and then that $C(S)$ is contained in that sphere as well.
4. Confirm the final formula for the optimal margin classifier at the end of the lecture.

References

[1] U.C. Irvine ML Repository. Auto MPG Dataset.Available at http://https://archive.ics.uci.edu/ml/datasets/Auto+MPG.

[2] Bertsekas, D. P. and Tsitsiklis, J. N. (2008). *Introduction to probability*. Athena Scientific.

[3] U.C. Irvine ML Repository. Sentiment Labelled Sentences Data Set.Available at https://archive.ics.uci.edu/ml/datasets/Sentiment+Labelled+Sentences.

[4] Jack Daoud. Marketing Analytics.Available at https://www.kaggle.com/datasets/jackdaoud/marketing-data.

[5] LeCun, Y., Cortes, C. and Burges, C. The MNIST Database.Available at http://yann.lecun.com/exdb/mnist/.

[6] Boser, B., Guyon, I. and Vapnik, V. A training algorithm for optimal margin classifiers. In *Colt '92: Proceedings of the fifth annual workshop on computational learning theory* (D. Haussler, ed) pp 144–52. ACM.

[7] Bennett, K. P. and Bredensteiner, E. J. (2000). Duality and geometry in SVM classifiers. In *Proceedings of the seventeenth international conference on machine learning* (P. Langley, ed). Morgan Kaufmann Publishers.

[8] Gorman, K. B., Williams, T. D. and Fraser, W. R. (2014). Ecological sexual dimorphism and environmental variability within a community of antarctic penguins (genus pygoscelis). *PLoS ONE* **9(3)** –13.

[9] Horst, A. Palmer penguins.Available at https://github.com/allisonhorst/palmerpenguins.

[10] Platt, J. C. (1998). *Sequential minimal optimization: A fast algorithm for training support vector machines*. Microsoft Research.

[11] Schölkopf, B., Burges, C. and Smola, A. (1998). *Advances in kernel methods: Support vector learning*. MIT Press.

A Lab for Linear Regression

A.0.1 Linear Algebra in Python/Numpy

In this lab we will use: - the `numpy` linear algebra package for computations - the `bokeh` plotting package for graphics

The next cell loads these libraries.

```
import numpy as np
from bokeh.plotting import figure
from bokeh.io import show, output_notebook
from bokeh.layouts import gridplot
output_notebook()
```

This routine will save typing later.

```
def comparison_plot(x,Y, Yhat):
    '''Plots Predicted vs True values for analysis of regression'''
    comparison_plot = figure(title='Difference between Measured and Predicted Values')
    comparison_plot.xaxis.axis_label='x'
    comparison_plot.yaxis.axis_label='Yhat-Y'
    comparison_plot.scatter(x=x,y=Yhat-Y)
    comparison_plot.line(x=[x.min(),x.max()],y=[0,0])
    return comparison_plot
```

A.0.2 Distance to a moving object (simulated data)

We will begin by looking at Figures 1 and 2 in the Linear Regression notes.

In Figure 1 we began with simulated data that represents the distance from an observer to a a moving object. We will generate that data. The observations occur at times $0, 1, 2, \ldots, 9$.

```
# Create a 1-d numpy array containing 0,1,...,9
x = np.array(range(10))
```

Let's suppose that the true velocity of the object is 15 m/s and the initial distance to the object is 150 m. Then the object's position is given by $d = 150 - 15x$ where x is time. However, when we measure the distance, there is a random error of between -10 and 10 meters. The function `np.random.uniform` returns uniformly distributed random numbers within a given range.

```
y = 150-15*x+np.random.uniform(-10,10,size=x.shape[0])
```

We will plot the data points (x, y).

```
f=figure(width=400,height=400,title='Measured Distance to Moving Object')
f.scatter(x=x,y=y)
f.xaxis.axis_label='time'
f.yaxis.axis_label='distance (m)'
show(f)
```

The points are roughly on a line, but not exactly, due to the error. The data matrix X for this problem consists of two columns, one of which is $0, 1, 2, \dots, 9$ and the other is $1, 1, 1, 1, \dots$. (See section 1.3 in the notes).

We make this by: - converting x to a column vector - creating a column vector of 1's - concatenating these

```
X=np.concatenate([x.reshape(-1,1),np.ones(shape=(x.shape[0],1))],axis=1)
```

```
X
```

The Y vector is our vector of measurements, except we need to make it a column vector.

```
Y = y.reshape(-1,1)
```

```
Y
```

Now we will use the formulae in equations (7) and (8) from the notes t compute the least squares line.

```
D = np.dot(X.transpose(),X)
```

The matrix M contains the slope and intercept of our least squares line.

```
M = np.dot(np.linalg.inv(D),np.dot(X.transpose(), Y))
print('Slope is {}, Intercept is {}'.format(M[0,0],M[1,0]))
```

The predicted values of Y are given by equation (8).

```
Yhat = np.dot(np.dot(np.dot(X,np.linalg.inv(D)),X.transpose()),Y)
```

Let's add these predicted values to our plot for comparison. Notice that the green dots lie along the regression line.

```
f.scatter(x=x,y=Yhat[:,0],color='green')
show(f)
```

We can connect the dots to see the line of best fit.

```
f.line(x=x,y=Yhat[:,0],color='green')
show(f)
```

```
Y.shape
Yhat.shape
```

```
show(comparison_plot(x,Y[:,0], Yhat[:,0]))
```

A.0.3 The MPG Data

Figure 2 in the notes shows data relating engine size to mileage for a group of cars. Your task is to reproduce the graph show in the figure.

A.0.3.1 Step 1. Load the data

The mileage data is in a comma separated file (csv) called `../data/auto-mpg/auto-mpg.csv` The command `np.genfromtxt` can be used to read in an array like this. You can examine the file by using the jupyter file browser and double clicking on the file name. You will see that: - the first row is a header - the last column is the type of car, which is a string; numpy can't handle that so it will set them to nan meaning 'not a number' - mpg is column zero - displacement is column 2

```
data = np.genfromtxt('../data/auto-mpg/auto-mpg.csv',delimiter=',',skip_header=1)
data
```

Now your job is to complete the code below, following the work we did above, so that you: - Create variable x that is just column 2 (displacement) of the data matrix - create variable

Y that is just column 0 (mpg) of the data matrix - Plot x and Y as a scatter plot. - Create a matrix whose first column is x and whose second column is all 1. - Compute the matrix $D = X^{\top}X$ - Find $M = D^{-1}X^{\top}Y$ - Find $Yhat = XD^{-1}X^{\top}Y$ - Plot $Yhat$ and the predicted line.

```
#x=data[]
#Y=data[]
#f = figure()
#f.scatter()
#show(f)
```

```

#X = np.concatenate([],axis=1)
#D = np.dot(...)
#M = np.dot(...)
#Yhat = ...
#f.scatter(...,color='green')
#f.line(...,color='green')
```

A.0.4 Residuals

One way to evaluate the fit of the line to the data is to compare Y and Yhat. Let's make a scatter plot of Y vs Yhat to see how they compare. If the fit is good, the points should cluster around the diagonal line y=x.

```
show(comparison_plot(x,Y,Yhat))
```

```
E = np.sum(np.square(Y-Yhat))
print('MSE={}'.format(E/Y.shape[0]))
```

Our MSE is 21.56.

Notice that there are a lot of cars with Yhat=30 mpg but whose true mpg (Y) are between 15 and almot 50 mpg. This means that, while engine displacement may be a good predictor of mileage for cars with relatively low mileage, among cars with higher mileage, something else must be going on.

A.0.5 Multivariate Data

For our first experiments with multivariate data, we will start with simulated data.

```
data = np.genfromtxt('../../data/multivar_simulated/data.csv',skip_header=1,delimiter=',')
```

Let's look at the data, just the first few rows.

```
data[:3,:]
```

As the README file says, the first column (column 0) is just the row number. Column 1 is the response variable Y, and columns 2 and 3 are the features

```
Y = data[:,1]
X = data[:,2:]
```

We append a column of ones to the data matrix.

Let's look at the relationship between the three columns. First, how does the response Y depend on the two columns of data? Plot Y vs X0 and X1 and show the result.

```
#f0 = figure()
#f0.xaxis.axis_label=
#f0.yaxis.axis_label =
#f0.scatter()
#f1 = figure()
#f1.xaxis.axis_label =
#f1.yaxis.axis_label =
#f1.scatter()
#show(gridplot())
```

It appears that increasing X0 increases Y, while increasing X1 decreases Y. What about the relationship between X0 and X1? Plot this as well

```
#f2 = figure()
#f2.xaxis.axis_label =
#f2.yaxis.axis_label =
#f2.scatter()
#show(f2)
```

The X0 and X1 variables seem independent of each other.

Now we can do the regression. First we add a column of 1's to our data matrix.

```
# create a column of ones
# append it to X
# check that X has shape 75 x 3
```

Now we compute the various matrices need for the computation.

```
#Compute the regression matrices D, Dinv, M, and Yhat
```

Notice that the coefficients of D are large (and those of Dinv) are small:

```
# look at D and Dinv
```

This can be a problem with accuracy, but we'll come back to it. The components of the model that we compute are:

```
# look at the M matrix
```

You should have gotten a result that says that Y is estimated as $\hat{y} = 1.78x_0 - 3.47x_1 + 6.06$ One way to visualize this is to plot the values of Yhat vs Y to see how close they are.

```
# plot the residuals using comparison_plot to compare Yhat and Y
```

As the plot shows, the points (y,yhat) cluster around the diagonal.

A.0.6 Covariance Matrix

Let's go back to look at the covariance matrix of the variables X0 and X1. This computed using centered coordinates.

```
# Center the data
#X0 = (X[] - X[].mean())
# then compute D0 and Cov = D0/X0.shape[0]
```

Notice that the off diagonal elements, which are the covariances of X0 and X1, are small in comparison to the diagonal elements. This reflects the fact that X0 and X1 are relatively (linearly) independent of each other.

A.0.7 The correlation coefficient

Pearson's R^2 is a quantitative measure of the strength of the linear relationship between two variables X and Y. It is computed as

$$R^2 = \frac{\sigma_{XY}^2}{\sigma_X^2 \sigma_Y^2}$$

From our matrix this is the offdiagonal element divided by the product of the diagonal elements. It it's close to 1, there is a strong relationship, if close to zero, there is not. In this case, the R^2 is essentially zero.

```
# compute the correlation coefficient.  Is it close to 1?  Why or why not?
```

A.0.8 Multivariate MPG data

Now let's refine our look at the MPG data by considering displacement, miles per gallon, and vehicle weight.
- column 0 is mpg - column 2 is displacement - column4 is vehicle weight

```
from itertools import combinations
```

```
data = np.genfromtxt('../data/auto-mpg/auto-mpg.csv',delimiter=',',skip_header=1)
```

```
# this code produces a grid of plots for you to study
d={}
d['mpg'] = data[:,0]
d['displacement'] = data[:,2]
d['weight'] = data[:,4]
d['horsepower'] = data[:,3]
figs = []
for x,y in combinations(['displacement','weight','mpg'],2):
    f = figure()
    f.xaxis.axis_label = x
    f.yaxis.axis_label = y
    f.scatter(x=d[x],y=d[y])
    figs.append(f)
g = gridplot([figs[0:2],[figs[2]]],plot_height=250,plot_width=250)
show(g)
```

The picture above is quite different from the simulated data because all three of the combinations show a relationship (remember, in the simulated data, the predictor variables didn't seem to be related to one another).

Another thing to observe is that the different data are on different size scales – vehicle weight is in the 1000's, displacmeent is in the 100's, and mpg is in the 10's.

```
# build the data matrix out of the corresponding columns and the variable Y from the mpg c
```

```
# compute D, Dinv, M and Yhat
```

```
# Look at the D matrix and its inverse and note the size of the coefficients
```

```
# plot the residuals of Y and Yhat using comparison_plot.  What do you see?
```

```
# here we compare the predicted and known values
show(comparison_plot(Y,Yhat))
```

The regression coefficients are:

```
M
```

```
# compute the MSE and compare the result to the one variable computation done earlier.  Is
```

Our MSE is a bit better in this multivariate case (18.4 vs 21 in the one-variable case).

Each unit increase in displacement reduces mileage by .016 and each unit increase in weight decreases mileage by -.0058. However, as the comparison plot shows, this relationship continues to breakdown for cars with high mileage – among cars that are predicted to have mileage around 30, the true mileage goes a lot higher, so some other factor must be at work.

A.0.9 The covariance matrix and correlation

To get a closer look at the relationship between weight and displacement let's compute the covariance matrix. To do that we need to center the variables.

```
# compute the covariance matrix from centered coordinates and then the correlation coeffic
# Note that they are highly correlated (R^2 is relatively close to 1).  How does this help
# regression doesn't add much to our ability to predict mpg.
```

B Principal Components Lab

In this lab we will apply Principal Components Analysis to the Auto-MPG dataset that we studied in the Chapter on LinearRegression. Before diving into the real data, we will work with the simulated data from the notes to show how to use python to and numpy to calculate the information we need.

```
import numpy as np
from bokeh.plotting import figure
from bokeh.io import output_notebook, show
from bokeh.models import ColumnDataSource, HoverTool
import seaborn as sns
from bokeh.palettes import Spectral10, Category10
output_notebook()
```

B.1 Simulated data for demo purposes

First we load the datamatrix.

```
data = np.genfromtxt('simulated_pca_data.csv',delimiter=',')
data.shape
secret_label = data[:,-1]
data=data[:,:-1]
```

The data consists of 200 samples, with 15 features per sample. It was constructed out of 4 different groups with different characteristics. These groups are labeled but we will ignore the labels for the moment. To carry out principal component analysis, we must:

1. center the data
2. compute the covariance matrix D
3. find the eigenvectors and eigenvalues of D

Then we can:

4. project the data into the space spanned by the first two eigenvectors of the covariance matrix and plot this

5. draw the loading axes on the plot.

B.1.1 Step 1. Centering the data

To center the data, we subtract the mean of each column from that column. The `mean` method computes the mean of each column of the data:

```
np.mean(data,axis=0)
```

```
data_centered = data - np.mean(data,axis=0)
```

Subtracting this from the data centers it – python understands that when you subtract a scalar from a column, you are really subtracting that scalar from every entry in the column.

Here we use a couple of programs we haven't discussed just to generate a gridplot for discussion.

```
sns.pairplot(pd.DataFrame(data_centered))
```

B.1.2 Step 2. Computing the Covariance Matrix

Remember that the covariance matrix is $X_0^\top X/N$.

```
D = np.dot(data_centered.transpose(),data_centered)/200
```

B.1.3 Correlation Coefficients

Remember that the correlation coefficient R^2 of two variables is

$$R^2_{XY} = \frac{\sigma^2_{XY}}{\sigma^2_X\sigma^2_Y}$$

We can extract this info from the covariance matrix.

```
def r(i,j):
    return D[i,j]/np.sqrt(D[i,i]*D[j,j])
```

```
for i in range(15):
    print(i, r(0,i))
```

B.1.4 Step 3. Finding the eigenvalues and eigenvectors of D

The command `np.linalg.eigh` returns a pair consisting of the vector of eigenvalues and the matrix of eigenvectors. By default, the eigenvalues are returned in increasing order, but we like them in decreasing order, so we reverse the list.

```
L, P = np.linalg.eigh(D)
L = L[::-1]
```

We can plot the eigenvalues.

```
eigenvalue_plot = figure(title='Eigenvalues of Covariance Matrix')
eigenvalue_plot.scatter(x=range(L.shape[0]),y=L,size=8)
eigenvalue_plot.line(x=range(L.shape[0]),y=L,color='red')
show(eigenvalue_plot)
```

B.1.5 Step 4. Projecting the data into the first two principal components

The columns of the matrix `P` are the eigenvectors, but they are ordered like the eigenvalues (from smallest to largest). So the two most significant principal components are the *last two* columns of the matrix, and we need to reverse their order.

```
PC2 = np.dot(data_centered,P[:,-2::-1])
```

```
scatter_plot = figure(title='Plot of First Two Principal Components',x_range=(-3,3),y_rang
scatter_plot.scatter(x=PC2[:,0],y=PC2[:,1])
show(scatter_plot)
```

```
colors=['red','green','blue','orange','black']
color_list = [colors[int(secret_label[i])] for i in range(200)]
scatter_plot = figure(title='Plot of First Two Principal Components with secret labels',x_
scatter_plot.scatter(x=PC2[:,0],y=PC2[:,1],color=color_list)
show(scatter_plot)
```

B.1.6 Step 4: Draw the loading directions

To project the axis of the i^{th} feature into the space spanned by the two principal eigenvectors, we draw a line in the direction of the vector we obtain by multiplying the row vector $[0, \dots, 1, \dots 0]$, where the 1 is in position i, into that space. But multiplying that vector times the matrix P just picks out the i^{th} of of P, so we want to draw a line in the direction of the point corresponding to the i^{th} row of P. For example, the 0^{th} feature is in the direction of $(PC[0,0], PC[0,1])$.

```
for i in range(PC2.shape[1]):
    scatter_plot.line(x=[-100*PC2[i,0],100*PC2[i,0]],y=[-100*PC2[i,1],100*PC2[i,1]],color=
scatter_plot.title.text = 'Plot of First Two Principal Components with Feature Loadings'
show(scatter_plot)
```

B.2 PCA for Auto Data

Let's look at what PCA can tell us about the auto data.

```
# we load the data file, and drop the rows with ? for the horsepower
data = np.genfromtxt('auto-mpg.csv',delimiter=',',skip_header=1)
```

In the section on linear regression we explored the relationship between the gas mileage and various other properties of each car model. We'll continue that analysis from the perspective of principal component analysis in this lab, focusing in particular on:

- mileage (mpg) (column 0)
- vehicle weight (column 4)
- acceleration (column 5) – note that big numbers mean poor acceleration!
- horsepower (column 3)
- displacement (column 2)

Display the data so you can see what it looks like.

Since we're only interested in mileage, weight, acceleration, and horsepower in this lab, let's just keep those features.

```
data = data[:,[0,2,3,4,5]]
data.shape
```

```
data[:4,:]
```

This array has some missing data in it, indicated by nan's (Not a Number). Here's how we find the bad spots.

```
np.argwhere(np.isnan(data))
```

This little function returns true if any element of a row of the matrix (axis=1) is nan. We look at it's first twenth entries.

```
np.isnan(data).any(axis=1)[:20]
```

argwhere looks for "True":

```
np.argwhere(np.isnan(data).any(axis=1))
```

```
good_rows = ~np.isnan(data).any(axis=1)
```

```
data = data[good_rows,:]
data.shape
```

Now let's go ahead and do PCA on this data.

Remember the steps:

1. Center the data (and rescale it)
2. Find the covariance matrix
3. Compute its eigenvalues and eigenvectors and plot the eigenvalues
4. Select the two largest eigenvalues and corresponding eigenvectors
5. Draw a scatter plot of the data projected into the span of these two principal directions
6. Draw the loadings.

```
# Step 1: center the data and rescale it
data_centered = data - np.mean(data,axis=0)
data_centered = data_centered/np.std(data,axis=0)
```

```
sns.pairplot(pd.DataFrame(data_centered))
```

```
# Step 2: Find the covariance matrix.  Hint: data.shape[0] is the number of samples
D = np.dot(data_centered.transpose(),data_centered)/data.shape[0]
```

```
D
```

```
# Step 3: Find the eigenvalues and eigenvectors and plot them
L, P = np.linalg.eigh(D)
L = L[::-1]
P = P[:,::-1]

eigenvalue_plot = figure(title='Eigenvalues')
eigenvalue_plot.circle(x=range(L.shape[0]),y=L)
eigenvalue_plot.line(x=range(L.shape[0]),y=L,color='green')
show(eigenvalue_plot)
```

```
# Step 4: Project and plot the data
PC2 = np.dot(data_centered, P[:,:2])
scatter_plot = figure(title='Principal Components')

scatter_plot.scatter(x=PC2[:,0],y=PC2[:,1])
show(scatter_plot)
```

```
# Step 5: add the loading directions
names = ['mpg','displacement','hp','weight','accel']
for i in range(5):
    scatter_plot.line(x=[0,P[i,0]],y=[0,P[i,1]],color=Category10[5][i],line_width=3,legend
scatter_plot.title.text = 'Principal Components with Loadings'
show(scatter_plot)
```

In looking at the figure above, notice that weight and mileage are almost perfect opposites – so there is an unavoidable tradeoff with higher weight vehicles having lower mileage. Moving to the lower right of the graph, you have better acceleration and also higher horsepower. Horsepower and displacement point in roughly the same direction, though not perfectly.

So:

- bottom left quadrant are bigger, relatively high mileage cars with poor acceleration
- bottom right quadrant are bigger, high-horsepower, slow cars
- upper right quadrant are bigger, faster, relatively high mileage cars
- upper left quadrant are smaller, faster, relatively high mileage cars.

```
names=[]
with open('auto-mpg.csv') as f:
    for line in f:
        fields = line.rstrip().split(',')
        names.append(fields[-1])
names = names[1:]
names = [names[i] for i in range(len(names)) if good_rows[i]]
```

```
source=ColumnDataSource({'pc0':PC2[:,0],'pc1':PC2[:,1],'type':names,'mpg':data[:,0],'disp'
```

```
scatter_plot=figure(title='PCA plot with labels (hover to see car type)')
scatter_plot.scatter(x='pc0',y='pc1',source=source)
scatter_plot.add_tools(HoverTool(tooltips=[("type","@type"),('mpg','@mpg'),('disp','@disp'
loadings = ['mpg','displacement','hp','weight','accel']
for i in range(5):
    scatter_plot.line(x=[0,P[i,0]],y=[0,P[i,1]],color=Category10[5][i],line_width=3,legend
show(scatter_plot)
```

B.3 Using sklearn

```
from sklearn.decomposition import PCA
```

```
P=PCA(n_components=2)
```

```
PC = P.fit_transform(data_centered)
```

```
PC.shape
```

```
scatter_plot=figure(title='sklearn version')
scatter_plot.scatter(x=PC[:,0],y=PC[:,1])
show(scatter_plot)
```

The loadings are in the components_ portion (each column is the projection of the corresponding feature into the space spanned by the PC).

```
P.components_
```

C A puzzle for you

The file `PenguinFeatures.csv` contains a subset of the data from the palmerpenguins website; each row is a set of measurements from a penguin. The separate file `PenguinLabels.csv` contains the species of the corresponding row.

Use PCA on the `PenguinFeatures.csv` to see if you can identify groups in the data that might be related to species. Can you construct a score or a pair of scores that could be used to distinguish penguins by species?

D Bayes Theorem Mini-Lab

```
from bokeh.plotting import figure
from bokeh.io import output_notebook,show
import numpy as np
output_notebook()
```

This lab is a chance to work with Bayes Theorem. The underlying dataset is a collection of SMS (text) messages that were labelled as either 'junk' or 'real' as part of an attempt to build a classifier that could filter out junk text messages.

The full dataset is in the 'complete.tsv' file – this is a "tab-separated" file, rather than a "comma-separated" file. But we won't be using this file directly. Instead, we will just work with the simpler file 'data.csv' which is a comma separated file with two columns. The first column is 0 or 1 depending on whether the corresponding text is real (0) or junk (1). The second column is the length of the associated text message.

```
data = np.genfromtxt('data.csv',delimiter=',',skip_header=1)
```

There are 5572 messages in the data.

```
data.shape
```

Let's separate out the junk and real messages to compare them. One way to do that is to create an index array. This command creates an array of True/False values based on whether that condition is true row-by-row.

```
data[:,0]==0
```

Notice that the first two entries in data are real and the third is junk:

```
data[:3,:]
```

Now we use our index array to extract the real rows.

```
real = data[data[:,0]==0,:]
```

```
real
```

There are 4825 real messages in the dataset.

```
real.shape
```

The average length of the real messages is computed like this:

```
real[:,1].mean(axis=0)
```

Now use a similar strategy to extract the junk rows and compute the mean length of the junk emails. What do you notice?

```
junk = data[data[:,0]==1,:]
print(junk.shape)
print('Mean Length of Junk Messages=',junk[:,1].mean(axis=0)
```

One way to use this information about the lengths is to set a threshold value, of say 100 characters, and divide the messages into “long” and “short” messages using this threshold. It seems that long messages are more likely to be junk. We can use Bayes theorem to try to quantify this.

Think of checking the length of a message like administering a test. Getting a positive result – finding a long message – should increase the odds that our message is junk.

From the point of view of Bayes Theorem, we are interested in

$$P(junk|long)$$

which we can compute as

$$P(junk|long) = \frac{P(long|junk)P(junk)}{P(long)}$$

And while we don’t really know these probabilities, we can estimate them by looking at the frequency counts in our data. (This approach is called “Naive” Bayes because we are naively assuming that the frequencies of data in our experiment are the real frequencies).

To get started, we need a 2x2 table of counts like this:

	long	short	total
junk			
real			
total			

from which we can compute the conditional probabilities.

These equations compute the number of elements in the (junk, long) and (junk, short) cells, with a threshold of 100 characters defining "Long".

```
junk_long = junk[junk[:,1]>=100].shape[0]
junk_short=junk[junk[:,1]<100].shape[0]
```

```
#real_long =
#real_short =
```

The conditional probability P(junk|long) is the percentage of long texts that are junk.

```
# P(junk|long) =
```

The probability of being junk unconditionally is about 13%.

```
(junk_long+junk_short)/5572
```

This function computes the conditional probability as a function of a threshold, which can vary.

```
def cp(threshold):
    junk_long = junk[junk[:,1]>=threshold].shape[0]
    real_long = real[real[:,1]>=threshold].shape[0]
    return junk_long/(junk_long+real_long)
```

Setting the threshold to 130 makes P(junk|long) maximal.

```
x=np.arange(200)
y=np.array([cp(i) for i in x])
f=figure()
f.line(x=x,y=y)
show(f)
```

Of course we are actually interested in detecting *real* messages. About 85% of our messages our real, so if we just say everything is real, we are right 85% of the time. Suppose we get a short message.

What is the probability that it is real?

E Naive Bayes Sentiment Analysis

E.1 Setup

In this lab we will take advantage of some parts of the scikit-learn (sklearn) machine learning library to carry out a Naive Bayes sentiment analysis of some amazon product reviews.

In addition to our usual libraries (numpy for linear algebra and bokeh for plotting) we load two functions from sklearn.

- `CountVectorizer` extracts wordcounts from documents
- `train_test_split` splits our data up into a "training set" from which we will derive our probabilities, and a "test" set that we will use to evaluate our classifier.

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import BernoulliNB
from sklearn.model_selection import train_test_split
from bokeh.models import ColumnDataSource, HoverTool
from bokeh.plotting import figure
from bokeh.io import show, output_notebook
import numpy as np
output_notebook()
```

First we read our data from the file. Each line of the file consists of a review, followed by a tab character, followed by a "0" or "1". We build a list of reviews and a corresponding list of labels from the file.

```
reviews = []
labels = []
with open("amazon.txt") as f:
    for line in f:
        review, label = line.strip().split('\t')
        reviews.append(review)
        labels.append(int(label))
```

The train_test_split breaks up the reviews and labels array randomly into two parts; by default, 75% of the data goes into the train set and 25% into the test set, though this is adjustable. Now we set aside the test data and work only with the training data until the end.

```
train_reviews, test_reviews, train_labels, test_labels = train_test_split(reviews, labels,
print('Length of train_reviews is ', len(train_reviews))
print('Length of test_reviews is ',len(test_reviews))
```

E.2 A simple example

Now we use the CountVectorizer function to analyze our reviews. The syntax here is that we create a CountVectorizer object and apply it to our review data. We set options to say that we only want to keep track of the 100 most common words in the reviews and (using `binary=True`)that we only want to mark words that occur with a zero or 1 – otherwise the routine will count the number of occurrences of each word.

```
F = CountVectorizer(max_features=100,binary=True)
```

To see how this works, let's apply it to a couple of simple sentences. The vectorizer expects a list of sentences, so we'll give a list of three sentences. It returns a matrix whose rows correspond to the sentences and whose columns are features corresponding to the words it discovered in the data.

```
simple = F.fit_transform(['This is a simple sentence that contains the word sentence twice
                          'This is another sentence.',
                          'A simple sentence has twice.'])
simple
```

The vectorizer returns a "sparse" array, which is an efficient way to store large matrices which are mostly zero. We'll see how to view it in a moment.

First, we can ask the vectorizer for the vocabulary that it uncovered. It returns a python dictionary that associates words to columns. So for example in this case the word `word` corresponds to the 9th column of the data matrix.

```
F.vocabulary_
```

Now the array `simple`.

```
simple.toarray()
```

Each row is the feature vector to a sentence, and each column corresponds to a word. So the first sentence does *not* contain the first key word (`another`) but the second one does.

The column sums tell us how often each word occurs (total) in the documents.

```
simple.sum(axis=0)
```

Suppose the first sentence is "positive" (labelled with 1) and the other two are negative (labelled with zero). The corresponding target array is Y.

```
Y = np.array([[1],[0],[0]])
Y
```

We can compute the frequencies in the positive documents.

```
Y.transpose() @ simple
```

and in the negative ones.

```
(1-Y).transpose() @ simple
```

The fourth word is 'sentence' which does indeed occur once in the type 1 sentences and 2 times in the type 0 ones.

The numbers of sentences of each type are $Y^T Y$ and $(1-Y)^T(1-Y)$ although numpy thinks these are two dimensional 1x1 arrays.

```
Y.transpose()@Y
```

```
(1-Y).transpose()@(1-Y)
```

We can compute the conditional probabilities with which each word occurs in the two types.

```
Pplus = Y.transpose()@simple/(Y.transpose()@Y)
Pplus
```

Just as a check, the first word, `another`, occurs only in the second sentence, so if has a 50% chance of occurring in a sentence labelled zero.

```
Pminus = (1-Y).transpose()@simple/((1-Y).transpose()@(1-Y))
Pminus
```

Now let's look at our training data. First we use the CountVectorizer to compute the feature matrix. We're going to add an option to tell the vectorizer to ignore elements of a list of "stop words" like he, his, at, him, … to simplify things.

E.3 The product review data

```
vectorizer = CountVectorizer(max_features=100,binary=True,stop_words='english')
```

```
train_matrix = vectorizer.fit_transform(train_reviews).toarray()
keywords = vectorizer.get_feature_names()
```

```
train_matrix.shape
```

Let's compute the frequencies of the 100 words in each of the two classes. We convert our labels into a numpy array.

```
train_y = np.array(train_labels)
```

```
train_y.shape
```

We use our formulae to compute the frequencies and the conditional probabilitiy vectors for the two classes.

```
freq_plus = (train_y.transpose()@train_matrix)
Nplus = train_y.transpose()@train_y
Pplus = freq_plus/Nplus
freq_minus = ((1-train_y).transpose()@train_matrix)
Nminus = ((1-train_y).transpose()@(1-train_y))
Pminus = freq_minus/Nminus
N = Nminus+Nplus
```

```
print(Nplus, Nminus)
```

Here we use a trick called "indirect sort" to find the words with largest P(w|+) and P(w|-). Argsort returns this *locations* of the elements in order. So indices[0] is the location in the Pplus array with the smallest value, indices[1] the next smallest value, and so on. We use these indices to extract the corresponding keywords.

```
indices = np.argsort(Pplus)
[keywords[i] for i in indices[::-1]][:20]
```

```
indices = np.argsort(Pminus)
[keywords[i] for i in indices[::-1]][:20]
```

```
# This is a fancy use of bokeh to add hover labels to the dots
source = ColumnDataSource({'+':Pplus,'-':Pminus,'word':keywords})
f=figure()
f.scatter(x='+',y='-',source=source)

f.xaxis.axis_label='P(w|+)'
f.yaxis.axis_label = 'P(w|-)'
f.line(x=[0,.2],y=[0,.2])
f.add_tools(HoverTool(tooltips=[("word","@word")]))
show(f)
```

To avoid taking the logarithm of zero, we increase all of the frequency counts by 1, as well as the Nplus and Nminus by 1. This is often called "smoothing."

```
freq_plus = freq_plus+1
freq_minus = freq_minus+1
Nplus = Nplus+2
Nminus = Nminus+2
```

```
LPplus = np.log(freq_plus/Nplus)
LPNplus = np.log(1-freq_plus/Nplus)
LPminus = np.log(freq_minus/Nminus)
LPNminus = np.log(1-freq_minus/Nminus)
```

Using the equation from the notes (which is essentially Bayes rule), we find:

```
posL = train_matrix @ LPplus + (1-train_matrix) @(LPNplus) - np.log(Nplus/(N+2))
negL  = train_matrix @ LPminus + (1-train_matrix)@(LPNminus) - np.log(Nminus/(N+2))
```

Recall that posL and negL are the likelihoods that a particular review is positive or negative, and our decision criterion is: - label 1 if posL-negL>0 - label 0 otherwise

Our decision array has a 1 if posL>negL and a zero otherwise.

```
decision = (posL > negL).astype(int)
```

Our check array as a 1 if decision and the original label agree, and zero otherwise.

```
check = (decision==train_y).astype(int)
```

```
np.sum(check)
```

They agree 591/750 times, or about 75% of the time. That's much better than guessing, which would only be right 50% of the time.

Finally, we use the test data to see if we can predict labels on "new" data. We re-use the LPplus and LPminus parameters, as well as the Nplus/N and Nminus/N from the training data. But we need to compute the data matrix for the test data *based on the features derived from the training data.*

```
vectorizer.fit(train_reviews)
test_matrix = vectorizer.transform(test_reviews).toarray()
test_y = np.array(test_labels)
```

```
test_matrix.shape
```

```
posL = test_matrix @ LPplus + (1-test_matrix)@(LPNplus) -np.log(Nplus/(Nplus+Nminus))
negL = test_matrix @ LPminus + (1-test_matrix)@(LPNminus) - np.log(Nminus/(Nplus+Nminus))
```

```
test_decision = (posL > negL).astype(int)
check = (test_decision == test_y).astype(int)
np.sum(check)
```

So we have correctly classified 182/250 reviews from the test set for an accuracy of 171/250 = 73%. Much better than guessing!

E.4 Using the sklearn facilities

The sklearn library can do all of this using built in routines. We add to the work above one more import.

```
from sklearn.naive_bayes import BernoulliNB, MultinomialNB
```

The BernoulliNB function takes the binary feature matrix and does all the computations associated with fitting the naive bayes model. Let's walk through it. We start with the train_reviews and train_labels lists. This cell builds the Bernoulli classifier B using the vectorized train_reviews and the train_labels data.

```
vectorizer = CountVectorizer(max_features=100,binary=True,stop_words='english')
V = vectorizer.fit(train_reviews)
X = V.transform(train_reviews)
train_y = np.array(train_labels)
B = BernoulliNB().fit(X,train_y)
```

This cell uses the fitted vectorizer to compute the data matrix for the test reviews.

```
T = V.transform(test_reviews)
test_y = np.array(test_labels)
```

Now we find the predictions using the matrix T.

```
predictions = B.predict(T)
```

```
predictions
```

The score method allows us to tell how well we did.

```
B.score(T,test_y)
```

The logs of the probabilities P(w|+/-) are stored inside B, and they agree with our computations.

```
B.feature_log_prob_
```

```
LPminus
```

```
LPplus
```

E.5 Your turn

Carry out the analysis above using the yelp and imdb data. You can use the sklearn facilities to make your life easier if you want. You can also try the multinomial classifier to see if it works better. In that case, you need to remove the binary=True flag from the vectorizer so that it counts frequencies. Here is an example of how the countvectorizer can compute term frequencies. You can also experiment with the "stop_words" flag.

```
vectorizer = CountVectorizer(max_features=10)
```

```
V = vectorizer.fit(["Here is a sentence", "Here is another sentence"])
```

```
V.vocabulary_
```

```
X = V.transform(["What are the frequencies in this here sentence", "I wrote a sentence abo
```

```
X.toarray()
```

```
V.get_feature_names()
```

F Multivariate normal distribution

```python
import numpy as np

from bokeh.plotting import figure, output_notebook, show
output_notebook()
```

```
Unable to display output for mime type(s): text/html
```

```
Unable to display output for mime type(s): application/javascript, application/vnd.bokehjs_lo
```

F.1 Univariate normal distribution

The normal distribution, also known as the Gaussian distribution, is defined by two parameters: the mean μ and the standard deviation σ. The square σ^2 of the standard deviation is called variance. We denote this distribution as:

$$\mathcal{N}(\mu, \sigma^2)$$

Given μ and σ, the probability density function is given by

$$p(x \mid \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

We call this distribution the univariate normal because it consists of only one random normal variable. Three examples of univariate normal distributions with different mean and variance are plotted in the next figure:

```python
def uni_normal(x, mean, variance):
    """pdf of the univariate normal distribution."""
    return ((1. / np.sqrt(2 * np.pi * variance)) *
            np.exp(-(x - mean)**2 / (2 * variance)))
```

F.1.1 Graphas of different univariate normal distributions

```
x = np.linspace(-4, 5, num=150)
fig = figure(plot_width=600, plot_height=400)
fig.line(x=x, y=uni_normal(x, mean=0, variance=1),color='red', legend_label='N(0,1)')
fig.line(x=x, y=uni_normal(x, mean=2, variance=2), color='blue', legend_label='N(2,2)')
fig.line(x=x, y=uni_normal(x, mean=-1, variance=0.5), color='black', legend_label='N(-1,0.
show(fig)
```

```
Unable to display output for mime type(s): text/html
```

```
Unable to display output for mime type(s): application/javascript, application/vnd.bokehjs_e
```

F.2 Multivariate normal distribution

The multivariate normal distribution is a multidimensional generalization of the univariae case. It represents the distribution of multiple random variables that can be correlated with each other.

Like the normal distribution, the multivariate normal is defined by sets of parameters: the mean vector   and the covariance matrix Σ which measures the mutual dependencies of the random variables.

The multivariate normal has a joint density given by:

$$p(\mathbf{x} \mid \ , \Sigma) = \frac{1}{\sqrt{(2\pi)^d |\Sigma|}} \exp\left(-\frac{1}{2}(\mathbf{x} - \ )^T \Sigma^{-1} (\mathbf{x} - \ )\right)$$

Where $\mathbf{x}$ a d-dimensional random vector,   is the mean vector, Σ is the covariance matrix of size $d \times d$, and $|\Sigma|$ its determinant. We denote this multivariate normal distribution as:

$$\mathcal{N}(\ , \Sigma)$$

```
def multi_normal(x, d, mean, covariance):
    """pdf of the multivariate normal distribution."""
    x_m = x - mean
    return (1. / (np.sqrt((2 * np.pi)**d * np.linalg.det(covariance))) *
            np.exp(-x_m.T@np.linalg.inv(covariance)@x_m / 2))
```

Examples of two bivariate normal distributions are plotted below.

The figure on the left is a bivariate distribution with the covariance between x_1 and x_2 set to 0 so that these 2 variables are independent:

$$\mathcal{N}\left(\begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}\right)$$

The figure on the right is a bivariate distribution with the covariance between x_1 and x_2 set to be different than 0 so that both variables are correlated. Increasing x_1 will increase the probability that x_2 will also increase:

$$\mathcal{N}\left(\begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}1 & 0.9\\0.9 & 1\end{bmatrix}\right)$$

```
def surface(mean, covariance):
    d=2
    nb_of_x = 200 # grid size
    x1s = np.linspace(-3, 3, num=nb_of_x)
    x2s = np.linspace(-3, 3, num=nb_of_x)
    x1, x2 = np.meshgrid(x1s, x2s)
    pdf = np.zeros((nb_of_x, nb_of_x))

    for i in range(nb_of_x):
        for j in range(nb_of_x):
            pdf[i,j] = multi_normal(
                np.matrix([[x1[i,j]], [x2[i,j]]]),
                d, mean, covariance)
    return x1, x2, pdf
```

```
%matplotlib notebook

import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import make_axes_locatable
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
```

```
fig, ax = plt.subplots(subplot_kw={"projection": "3d"})

mean = np.matrix([[0.], [0.]])
```

```
covariance = np.matrix([
    [1., 0.],
    [0., 1.]])
x, y, z = surface(mean, covariance)

surf = ax.plot_surface(x, y, z, cmap=cm.coolwarm,
                       linewidth=0, antialiased=False)

ax.set_zlim(0, 0.2)
fig.colorbar(surf, shrink=0.5, aspect=5)

plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
fig, ax = plt.subplots(subplot_kw={"projection": "3d"})

mean = np.matrix([[0.], [0.]])
covariance = np.matrix([
    [1., 0.9],
    [0.9, 1.]])
x, y, z = surface(mean, covariance)

surf = ax.plot_surface(x, y, z, cmap=cm.coolwarm,
                       linewidth=0, antialiased=False)

ax.set_zlim(0, 0.3)
fig.colorbar(surf, shrink=0.5, aspect=5)

plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

F.3 Linear Regression Revisited

```
### Generating the data ##########
n_points=15
P=[]
for i in range (0,n_points):
    x=np.random.random()
    y=np.sin(2*np.pi*x)
    z=y+np.random.normal(0,0.05)
#   print (x,z)
    P.append([x,z])

plt.figure()
plt.scatter(*zip(*P))
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
P
```

```
[[0.45606998421703593, 0.20958138658243655],
 [0.5142344384136116, -0.02954637874443599],
 [0.5924145688620425, -0.5363521780014958],
 [0.17052412368729153, 0.8263448541638341],
 [0.06505159298527952, 0.33741999169677145],
 [0.09767211400638387, 0.6706336646478746],
 [0.6842330265121569, -1.0179853258193641],
 [0.4951769101112702, 0.014124190737620747],
 [0.034388521115218396, 0.19144708871067084],
 [0.662522284353982, -0.8137784232084905]]
```

```
### Result ##########

P=[[0.8846440661991892, -0.864791215635069],
  [0.793349886821054, -1.32738612014193],
  [0.7354408415584879, -1.18222466237236],
  [0.42187176484679356, 0.304255805886633],
  [0.011883272993057137, 0.101594120287724],
  [0.22677018897293355, 1.13377458999431],
  [0.9785306716286032, -0.147028527196347],
  [0.043107697015664526, 0.247622971933151],
  [0.8900032869307322, -0.605625802202937],
  [0.8883627996249959, -0.649537521948140]]
```

```
X1=np.empty((2,n_points))
for i in range (0,2):
    for j in range(0,n_points):
        X1[i,j]=(P[j][0])**i


X3=np.empty((4,n_points))
for i in range (0,4):
    for j in range(0,n_points):
        X3[i,j]=(P[j][0])**i

X6=np.empty((7,n_points))
for i in range (0,7):
    for j in range(0,n_points):
        X6[i,j]=(P[j][0])**i

X9=np.empty((10,n_points))
for i in range (0,10):
    for j in range(0,n_points):
        X9[i,j]=(P[j][0])**i

T=np.empty((n_points,1))
for i in range (0,n_points):
    T[i,0]=P[i][1]
```

```
Y1=X1@X1.transpose()
Y3=X3@X3.transpose()
Y6=X6@X6.transpose()
Y9=X9@X9.transpose()

W1=np.linalg.inv(Y1)@X1@T
```

```
W3=np.linalg.inv(Y3)@X3@T
W6=np.linalg.inv(Y6)@X6@T
W9=np.linalg.inv(Y9)@X9@T
```

```
def F1(x):
    return W1[0][0]+W1[1][0]*x

def F3(x):
    return W3[0][0]+W3[1][0]*x+W3[2][0]*x**2+W3[3][0]*x**3

def F6(x):
    return W6[0][0]+W6[1][0]*x+W6[2][0]*x**2+W6[3][0]*x**3+W6[4][0]*x**4+W6[5][0]*x**5+W6[

def F9(x):
    return W9[0][0]+W9[1][0]*x+W9[2][0]*x**2+W9[3][0]*x**3+W9[4][0]*x**4+W9[5][0]*x**5+W9[
```

```
xe=np.arange(0,1,0.02)

plt.figure()
plt.plot(xe, F1(xe),'k')
plt.plot(xe, F3(xe),'k')
plt.scatter(*zip(*P))


plt.figure()
plt.plot(xe, F6(xe),'k')
plt.scatter(*zip(*P))

plt.figure()
plt.plot(xe, F9(xe),'k')
plt.scatter(*zip(*P))
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
<matplotlib.collections.PathCollection at 0x7f902a62a510>
```

```
print(F6(0.123))
print(F9(0.123))
```

```
0.5486579835460237
24.875366088108674
```

```
alpha=0.01
beta=1000

X9=np.empty((10,n_points))
for i in range (0,10):
    for j in range(0,n_points):
        X9[i,j]=(P[j][0])**i

T=np.empty((n_points,1))
for i in range (0,n_points):
    T[i,0]=P[i][1]

Y9=X9@X9.transpose()

S_inv=beta*Y9+alpha*np.identity(10)

S=np.linalg.inv(S_inv)

W9=beta*S@X9@T


def FB9(x):
    return W9[0][0]+W9[1][0]*x+W9[2][0]*x**2+W9[3][0]*x**3 \
           +W9[4][0]*x**4+W9[5][0]*x**5+W9[6][0]*x**6 \
           +W9[7][0]*x**7+W9[8][0]*x**8+W9[9][0]*x**9
```

```
xe=np.arange(0,1,0.02)

plt.figure()
plt.plot(xe, FB9(xe),'k')
plt.scatter(*zip(*P))
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
<matplotlib.collections.PathCollection at 0x7f90401da7d0>
```

G Libraries

```python
import numpy as np
rng =np.random.default_rng()
np.set_printoptions(suppress=True, precision=4)
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import OneHotEncoder
from scipy.special import logsumexp
from bokeh.plotting import figure, output_notebook, show
output_notebook()
from tqdm import tqdm
```

```
Unable to display output for mime type(s): text/html

Unable to display output for mime type(s): application/javascript, application/vnd.bokehjs_lo
```

G.1 Multiclass Regression

G.1.1 Simulate data

We will simulate some tiny image data, where each image has is an array of five 0-255 values and there are 4 different images: x0x0x 00xxx xxx00 x000x

```python
samples=[60,40,80,120]
variability=40

x0=[30,200,30,200,30]
x1=[200,200,30,30,30]
x2=[30,30,30,200,200]
x3=[30,200,200,200,30]

data = np.concatenate([rng.normal(loc=x,scale=variability,size=(n,5)).astype(int) for n,x

# cut off entries bigger than 255
```

```
labels = np.array([0]*samples[0]+[1]*samples[1]+[2]*samples[2]+[3]*samples[3])

# rearrange the rows
rows = list(range(sum(samples)))
shuffled = rng.shuffle(rows)
data = data[rows,:]
labels=labels[rows]
```

```
labels
```

```
array([3, 1, 2, 0, 2, 3, 2, 3, 3, 3, 3, 0, 1, 2, 3, 3, 2, 2, 1, 3, 3, 3,
       2, 2, 2, 2, 3, 1, 3, 2, 3, 0, 0, 0, 2, 3, 1, 3, 2, 0, 3, 2, 3, 0,
       3, 3, 2, 2, 0, 2, 0, 3, 2, 3, 1, 0, 1, 0, 1, 0, 3, 1, 2, 2, 3, 2,
       2, 3, 1, 2, 3, 1, 3, 2, 3, 0, 3, 3, 3, 1, 2, 0, 2, 3, 3, 3, 0, 1,
       2, 3, 2, 3, 2, 3, 3, 1, 3, 3, 0, 2, 3, 3, 0, 2, 0, 1, 3, 1, 0, 3,
       1, 3, 3, 0, 2, 0, 1, 1, 2, 0, 2, 2, 2, 2, 3, 3, 3, 2, 2, 1, 3, 3,
       2, 0, 2, 0, 3, 0, 3, 2, 0, 1, 0, 3, 3, 3, 2, 2, 3, 0, 3, 3, 0, 1,
       3, 2, 0, 1, 2, 2, 1, 2, 0, 3, 2, 3, 2, 2, 3, 2, 3, 3, 0, 2, 1, 2,
       2, 3, 3, 3, 0, 1, 3, 3, 3, 0, 0, 2, 3, 0, 3, 0, 3, 0, 3, 3, 2, 2,
       0, 0, 3, 0, 0, 0, 3, 2, 0, 2, 3, 3, 3, 3, 3, 1, 3, 2, 3, 1, 2, 0,
       1, 0, 0, 1, 2, 3, 3, 3, 2, 0, 3, 3, 2, 2, 1, 2, 0, 0, 0, 3, 3, 3,
       0, 1, 1, 2, 2, 3, 2, 0, 3, 1, 3, 3, 3, 0, 3, 1, 3, 3, 3, 3, 3, 2,
       3, 3, 3, 1, 2, 0, 3, 3, 2, 3, 0, 3, 1, 1, 2, 3, 2, 3, 0, 3, 3, 3,
       1, 0, 1, 2, 0, 3, 2, 2, 3, 0, 2, 3, 2, 3])
```

```
### Test the sklearn routines

L=LogisticRegression(solver='liblinear')

#### add the constant feature
data = (data-data.mean(axis=0))/data.std(axis=0)
L.fit(data,labels)
print("The coefficient matrix")
print(L.coef_)
print("The intercepts")
print(L.intercept_)
```

```
The coefficient matrix
[[-0.7448  1.4407 -3.0763  1.2578 -1.5464]
```

```
 [ 1.6059  0.2207 -0.3796 -1.8167 -0.3021]
 [-0.2022 -2.2679 -0.5316  0.2552  2.3482]
 [-0.3142  0.7471  4.2188  0.5531 -0.6291]]
The intercepts
[-2.7463 -3.6326 -2.4263 -1.3264]
```

```
print("Accuracy on the test data is {} percent".format(100*L.score(data,labels)))
```

```
Accuracy on the test data is 99.0 percent
```

```
def sigma(x,m):
    """computes the softmax function on the rows of the data matrix x for the weights m"""
    y = x @ m
    j = np.exp(y)
    p = j/j.sum(axis=1,keepdims=True)
    return p
# Use sklearn to create a "one-hot" encoding of the labels

E=OneHotEncoder()
Y=E.fit_transform(labels.reshape(-1,1)).toarray()

def descent(x,y, max_iter=10000,nu=.000001,M=None):
    """does gradient ascent to maximum likelihood for data=x and labels (one-hot)=y"""
    features = x.shape[1]
    classes = y.shape[1]
    grads=[]
    if M is None:
        M = rng.normal(loc=0,scale=1,size=(features,classes))
    for i in tqdm(range(max_iter)):
        P=sigma(x,M)
        grad = data.transpose() @ (Y-P)
        M = M+nu*grad
        grads.append(np.max(np.abs(grad)))

    return M,grads
```

```
M,g=descent(data,Y,max_iter=10000,nu=1e-5)
print(M)
```

```
100%|          | 10000/10000 [00:00<00:00, 34120.70it/s]
```

```
[[ 0.312   2.2551  0.0543 -0.1579]
 [-0.0391 -1.282  -2.323  -0.3414]
 [-1.993  -0.7251 -0.9839  2.2387]
 [ 1.1905  0.005   0.8452  0.7962]
 [-1.095  -0.8986  0.8361 -1.2064]]
```

```
### the predicted label is the one with maximum probability
predictions=np.argmax(sigma(data,M),axis=1)

### accuracy compares predicted to true
correct=((predictions==labels).sum())/predictions.shape[0]

print("Accuracy here is {} percent".format(100* correct))
```

```
Accuracy here is 98.33333333333333 percent
```

```
f=figure(title='Size of largest coefficient in gradient vs. number of iterations')
f.line(x=list(range(len(g))),y=g)
show(f)
```

```
Unable to display output for mime type(s): text/html
```

```
Unable to display output for mime type(s): application/javascript, application/vnd.bokehjs_e
```

```
np.min(g)
```

```
8.84359279700982
```

```
M,g=descent(data,Y,max_iter=100000,nu=1e-5)
```

```
100%|          | 100000/100000 [00:02<00:00, 34717.29it/s]
```

```python
f=figure(title='Size of largest coefficient in gradient vs. number of iterations')
f.line(x=list(range(len(g))),y=g)
show(f)
```

```
Unable to display output for mime type(s): text/html
```

```
Unable to display output for mime type(s): application/javascript, application/vnd.bokehjs_e
```

```python
g[-1]
```

```
1.0148803989102506
```

```python
M,g=descent(data,Y,max_iter=10000,nu=1e-5)
bigg=[]
for i in range(50):
    M,g=descent(data,Y,max_iter=10000,nu=(1e-5)/2,M=M)
    bigg = bigg + g
    print(g[-1])
```

```
100%|      | 10000/10000 [00:00<00:00, 31629.11it/s]
100%|      | 10000/10000 [00:00<00:00, 34738.17it/s]
```

```
5.239145689147484
```

```
100%|      | 10000/10000 [00:00<00:00, 34483.44it/s]
```

```
3.9878568557014606
```

```
100%|      | 10000/10000 [00:00<00:00, 35510.37it/s]
```

```
3.2370838044507746
```

```
100%|      | 10000/10000 [00:00<00:00, 35127.73it/s]
```

```
2.7618220273640244

100%|      | 10000/10000 [00:00<00:00, 34602.99it/s]

2.417622645230818

100%|      | 10000/10000 [00:00<00:00, 35075.82it/s]

2.1562056444175015

100%|      | 10000/10000 [00:00<00:00, 35889.06it/s]

1.9505204967695517

100%|      | 10000/10000 [00:00<00:00, 34017.56it/s]

1.7842047392362406

100%|      | 10000/10000 [00:00<00:00, 35588.83it/s]

1.6467654804026377

100%|      | 10000/10000 [00:00<00:00, 35080.20it/s]

1.5311563800187553

100%|      | 10000/10000 [00:00<00:00, 34496.66it/s]

1.43246554273079

100%|      | 10000/10000 [00:00<00:00, 34886.48it/s]

1.3471615772581536

100%|      | 10000/10000 [00:00<00:00, 31951.29it/s]

1.27263878965863
```

```
100%|      | 10000/10000 [00:00<00:00, 35169.68it/s]

1.206931444571136

100%|      | 10000/10000 [00:00<00:00, 34542.03it/s]

1.1485279437627884

100%|      | 10000/10000 [00:00<00:00, 34223.60it/s]

1.0962463437324548

100%|      | 10000/10000 [00:00<00:00, 34882.36it/s]

1.0491487854491288

100%|      | 10000/10000 [00:00<00:00, 33995.12it/s]

1.0064813257026652

100%|      | 10000/10000 [00:00<00:00, 35182.01it/s]

0.9676307733987306

100%|      | 10000/10000 [00:00<00:00, 34914.21it/s]

0.9320931668975794

100%|      | 10000/10000 [00:00<00:00, 35915.98it/s]

0.8994503810866581

100%|      | 10000/10000 [00:00<00:00, 35561.07it/s]

0.8693525147731155

100%|      | 10000/10000 [00:00<00:00, 35689.16it/s]
```

```
0.8415044551252755

100%|      | 10000/10000 [00:00<00:00, 32756.48it/s]

0.8156555053762804

100%|      | 10000/10000 [00:00<00:00, 34761.26it/s]

0.7915912893861966

100%|      | 10000/10000 [00:00<00:00, 32572.49it/s]

0.769127369513328

100%|      | 10000/10000 [00:00<00:00, 33165.34it/s]

0.7481041684163334

100%|      | 10000/10000 [00:00<00:00, 33007.30it/s]

0.7283828936511145

100%|      | 10000/10000 [00:00<00:00, 31578.88it/s]

0.7098422409711951

100%|      | 10000/10000 [00:00<00:00, 34350.23it/s]

0.6923757077739388

100%|      | 10000/10000 [00:00<00:00, 34952.53it/s]

0.6758893886374271

100%|      | 10000/10000 [00:00<00:00, 34262.23it/s]

0.6603001547529692
```

```
100%|      | 10000/10000 [00:00<00:00, 34585.07it/s]

0.6455341413016176

100%|      | 10000/10000 [00:00<00:00, 34083.82it/s]

0.6315254835463484

100%|      | 10000/10000 [00:00<00:00, 34610.32it/s]

0.6182152551008164

100%|      | 10000/10000 [00:00<00:00, 34106.83it/s]

0.6055505715424204

100%|      | 10000/10000 [00:00<00:00, 34464.91it/s]

0.5934838300222136

100%|      | 10000/10000 [00:00<00:00, 34865.89it/s]

0.5819720613386885

100%|      | 10000/10000 [00:00<00:00, 34017.34it/s]

0.570976375491053

100%|      | 10000/10000 [00:00<00:00, 34391.47it/s]

0.56046148530921

100%|      | 10000/10000 [00:00<00:00, 33111.30it/s]

0.5503952955957444

100%|      | 10000/10000 [00:00<00:00, 34807.88it/s]
```

```
0.5407485474780324

100%|      | 10000/10000 [00:00<00:00, 35055.80it/s]

0.5314945094822844

100%|      | 10000/10000 [00:00<00:00, 35515.36it/s]

0.5226087083028782

100%|      | 10000/10000 [00:00<00:00, 35320.13it/s]

0.514068693424929

100%|      | 10000/10000 [00:00<00:00, 35421.01it/s]

0.505853830722268

100%|      | 10000/10000 [00:00<00:00, 33214.08it/s]

0.49794512094101095

100%|      | 10000/10000 [00:00<00:00, 34978.59it/s]

0.49032503962727825

100%|      | 10000/10000 [00:00<00:00, 34968.82it/s]

0.48297739559155944

100%|      | 10000/10000 [00:00<00:00, 34965.30it/s]

0.4758872054455315
```

```
M
```

```
array([[-1.3558,  1.4395, -0.5234, -0.7165],
       [ 2.073 ,  0.6709, -2.088 ,  2.2809],
       [-4.5633,  0.0461, -0.9698,  5.4576],
       [ 1.6446, -1.8624,  0.5536,  0.871 ],
       [-1.89  ,  0.1597,  3.23  , -1.538 ]])
```

H SVM Lab

In this lab we will work with the sklearn SVM Library to apply support vector machines to the penguin data set.

We load the usual libraries: - numpy - bokeh - the sklearn library which stands for support vector classifier

```
import numpy as np
from numpy.random import default_rng
from bokeh.io import output_notebook, show
from bokeh.plotting import figure
from bokeh.models import CategoricalColorMapper
from sklearn.svm import SVC
rng = default_rng(5)
output_notebook()
```

The following two functions are utilities that will help us to extract information from the SVC object in a form suitable for plotting. The key is that the vector w that gives the optimal margin is $\sum t_i \alpha_i x_i$ where the x_i are the "support vectors" in `P.support_vectors_`, the t_i are the associated labels, and the α_i are the "dual coefficients" in `P.dual_coef_`. See the documentation of the mathematical formulation, section 1.4.7.1.

```
def hyperplane(P,x,z=0):
    """Given an SVC object P and an array of vectors x, computes the hyperplane wx+b=z"""
    alphas = P.dual_coef_
    svs = P.support_vectors_
    c = P.intercept_[0]-z
    a = np.sum(alphas.T*svs,axis=0)[0]
    b = np.sum(alphas.T*svs,axis=0)[1]
    return (-c-a*x)/b

def pts(P):
    """Given an SVC object P, returns the two closest points in the associated reduced con
    alphas = P.dual_coef_[0]
    svs = P.support_vectors_
    plus_indices = np.where(alphas>0)
    minus_indices = np.where(alphas<=0)
    alphas = alphas.reshape(-1,1)
    pluspt = np.sum(alphas[plus_indices]*svs[plus_indices],axis=0)/np.sum(alphas[plus_indi
    minuspt = np.sum(alphas[minus_indices]*svs[minus_indices],axis=0)/np.sum(alphas[minus_
    return pluspt, minuspt
```

H.1 Simulated data illustration

The command

~~~
computes the Support Vector Classifier.  Here C bounds the coefficients $$\lambda_{i}$$, a
linearly separable you should you should take C large to find the optimal margin based on
read the "mathematical formulation" part of the SVC documentation to see the exact corresp

::: {.cell}
``` {.python .cell-code}
N=200
# Generate two random point sets and plot them
A = rng.normal(-1,.3,size=(N,2))
B = rng.normal(1,.3,size=(N,2))

f=figure(x_range=[-2,2],y_range=[-2,2],height=500,width=500)

f.scatter(x=A[:,0],y=A[:,1],color='blue')
f.scatter(x=B[:,0],y=B[:,1],color='green')

data  = np.concatenate([A,B],axis=0)
labels = np.array([0]*N+[1]*N)

# compute the svc classifier

P = SVC(kernel='linear',C=1000).fit(data,labels)

# draw the separating hyperplane, plus the hyperplanes that define the margin.
# here we are extracting information from the SVC object.

x=np.linspace(-2,2,100)
y = hyperplane(P,x)
f.line(x=x,y=y)
~~~

```
y = hyperplane(P,x,1)
f.line(x=x,y=y)
y = hyperplane(P,x,-1)
f.line(x=x,y=y)
show(f)
```

:::

```
# P.support_vectors_ gives the points that lie on the marginal hyperplanes.
P.support_vectors_
```

```
# P.dual_coef_ gives the associated lambda's multiplied by +/-1 depending on the label
P.dual_coef_
```

```
# To find the "closest points" use the pts function
pts(P)
xs = [pts(P)[0][0],pts(P)[1][0]]
ys = [pts(P)[0][1],pts(P)[1][1]]
```

```
f.line(x=xs,y=ys,color='black',line_width=5)
f.scatter(x=xs,y=ys,color='black',size=8)
show(f)
```

H.2 Penguins and multiclass classification

I've simplified life a little by removing missing data from the penguin data set and by restricting to numerical features.

There are more than two features for the penguin data, but lets start by looking at the two we considered in the theoretical discussion: culmen length and body mass. These are features 0 and 3.

```
data = np.genfromtxt("penguin_data.csv",delimiter=',',skip_header=1)
labels = np.genfromtxt("penguin_labels.csv",delimiter=',',dtype=int,skip_header=1)
working_data=data[:,[0,3]]
working_data[:,1] = working_data[:,1]/200
colors = ['red','blue','green']
penguin_colors = np.array([colors[i] for i in labels])
```

```
f=figure(title='Penguin Data: culmen length vs body mass/200',x_range=[30,60],y_range=[10,
f.scatter(x=working_data[:,0],y=working_data[:,1],color=penguin_colors)
show(f)
```

We can try “one vs rest” classification where we consider each group against all of the other points.

```
red_points = np.where(labels==0)
blue_points =np.where(labels==1)
green_points = np.where(labels ==2)
```

```
blue_vs_others = np.array([0 if x ==1 else 1 for x in labels])
red_vs_others = np.array([0 if x==0 else 1 for x in labels])
green_vs_others = np.array([0 if x==2 else 1 for x in labels])
```

```
Pred = SVC(kernel='linear',C=1000).fit(working_data,red_vs_others)
Pgreen = SVC(kernel='linear',C=1000).fit(working_data,green_vs_others)
Pblue = SVC(kernel='linear',C=1000).fit(working_data,blue_vs_others)
```

```
f=figure(title='Penguin Data: culmen length vs body mass/200',x_range=[30,60],y_range=[10,
f.scatter(x=working_data[:,0],y=working_data[:,1],color=penguin_colors)
x=np.linspace(30,60,100)
yred=hyperplane(Pred,x)
y0 = hyperplane(Pred,x,1)
y1 = hyperplane(Pred,x,-1)
f.line(x=x,y=yred,line_width=3,color='black',line_dash='dashed',legend_label='red vs other
f.line(x=x,y=y0,line_width=1,alpha=.5,color='black',line_dash='dashed',legend_label='red v
f.line(x=x,y=y1,line_width=1,alpha=.5,color='black',line_dash='dashed',legend_label='red v

#f.line(x=x,y=yblue,line_width=3,color='black',line_dash='dotted',legend_label='blue vs ot
#f.line(x=x,y=ygreen,line_width=3,color='black',line_dash='dotdash',legend_label='green vs
show(f)
```

```
f=figure(title='Penguin Data: culmen length vs body mass/200',x_range=[30,60],y_range=[10,
f.scatter(x=working_data[:,0],y=working_data[:,1],color=penguin_colors)
x=np.linspace(30,60,100)
yred=hyperplane(Pred,x)
y0 = hyperplane(Pred,x,1)
y1 = hyperplane(Pred,x,-1)
```

```
f.line(x=x,y=yred,line_width=3,color='black',line_dash='dashed',legend_label='red vs other
f.line(x=x,y=y0,line_width=1,alpha=.5,color='black',line_dash='dashed',legend_label='red v
f.line(x=x,y=y1,line_width=1,alpha=.5,color='black',line_dash='dashed',legend_label='red v
yblue = hyperplane(Pblue,x)
y0 = hyperplane(Pblue,x,1)
y1 = hyperplane(Pblue,x,-1)
f.line(x=x,y=yblue,line_width=3,color='black',line_dash='dotted',legend_label='blue vs oth
f.line(x=x,y=y0,line_width=1,alpha=.5,color='black',line_dash='dotted',legend_label='blue
f.line(x=x,y=y1,line_width=1,alpha=.5,color='black',line_dash='dotted',legend_label='blue
ygreen=hyperplane(Pgreen,x)
y0 = hyperplane(Pgreen,x,1)
y1 = hyperplane(Pgreen,x,-1)
f.line(x=x,y=ygreen,line_width=3,color='black',line_dash='dotdash',legend_label='green vs
f.line(x=x,y=y0,line_width=1,alpha=.5,color='black',line_dash='dotdash',legend_label='gree
f.line(x=x,y=y1,line_width=1,alpha=.5,color='black',line_dash='dotdash',legend_label='gree
show(f)
```

The SVC classifier computes a decision function by evaluating the different hyperplane functions. So it looks at fred(p), fblue(p), and fgreen(p). It assigns the point to the class where the value is largest. Choose C as large as possible to use the most straightforward version of the problem.

```
Pall = SVC(kernel='linear',C=1000).fit(working_data,labels)
Pall.predict(working_data)
predicted_colors = [colors[i] for i in Pall.predict(working_data)]
f=figure(title='predicted classification',x_range=[30,60],y_range=[10,35])
f.scatter(x=working_data[:,0],y=working_data[:,1],color=predicted_colors)
yred=hyperplane(Pred,x)
y0 = hyperplane(Pred,x,1)
y1 = hyperplane(Pred,x,-1)
f.line(x=x,y=yred,line_width=3,color='black',line_dash='dashed',legend_label='red vs other
f.line(x=x,y=y0,line_width=1,alpha=.5,color='black',line_dash='dashed',legend_label='red v
f.line(x=x,y=y1,line_width=1,alpha=.5,color='gray',line_dash='dashed',legend_label='red vs
yblue = hyperplane(Pblue,x)
y0 = hyperplane(Pblue,x,1)
y1 = hyperplane(Pblue,x,-1)
f.line(x=x,y=yblue,line_width=3,color='black',line_dash='dotted',legend_label='blue vs oth
f.line(x=x,y=y0,line_width=1,alpha=.5,color='black',line_dash='dotted',legend_label='blue
f.line(x=x,y=y1,line_width=1,alpha=.5,color='gray',line_dash='dotted',legend_label='blue v
ygreen=hyperplane(Pgreen,x)
y0 = hyperplane(Pgreen,x,1)
```

```
y1 = hyperplane(Pgreen,x,-1)
f.line(x=x,y=ygreen,line_width=3,color='black',line_dash='dotdash',legend_label='green vs
f.line(x=x,y=y0,line_width=1,alpha=.5,color='black',line_dash='dotdash',legend_label='gree
f.line(x=x,y=y1,line_width=1,alpha=.5,color='gray',line_dash='dotdash',legend_label='green
show(f)
```

```
print('Classifier yields accuracy of {:2f}%'.format(Pall.score(working_data,labels)))
```

H.3 Using all the features

We can train an SVC classifier on 25% of the data and still get exceptionally good predictions using all the features.

```
from sklearn.model_selection import train_test_split
data_train, data_test, labels_train, labels_test = train_test_split(data,labels,test_size=
P = SVC(kernel='linear').fit(data_train,labels_train)
```

```
P.score(data_test,labels_test)
```

I f-MNIST with SVM

The two files fmnist_data.csv and fmnist_labels.csv are a selection of 10000 records of the F-MNIST data set, slightly cleaned up, for use in SVC classification.

The fmnist_data.csv files contains the 10000x784 records of the images, and the labels contains the 10000 associated labels 0-9.

```
data = np.genfromtxt('fmnist_data.csv',delimiter=',')
labels=np.genfromtxt('fmnist_labels.csv',delimiter=',')
```

```
data_train, data_test, labels_train, labels_test = train_test_split(data,labels,test_size=
```

```
#P = SVC().fit(data_train,labels_train)
```

J Classifying MNIST via Neural Networks

The **MNIST** (Modified National Institute of Standards and Technology) database is a large database of handwritten digits that is commonly used for training various image processing systems.

Before we can build any neural networks we need to import a few things from Keras and prepare our data. The following code extracts the MNIST dataset, provided by Keras, and flattens the 28x28 pixel images into a vector with length 784. Additionally, it modifies the labels from a numeric value 0-9 to a one-hot encoded vector.

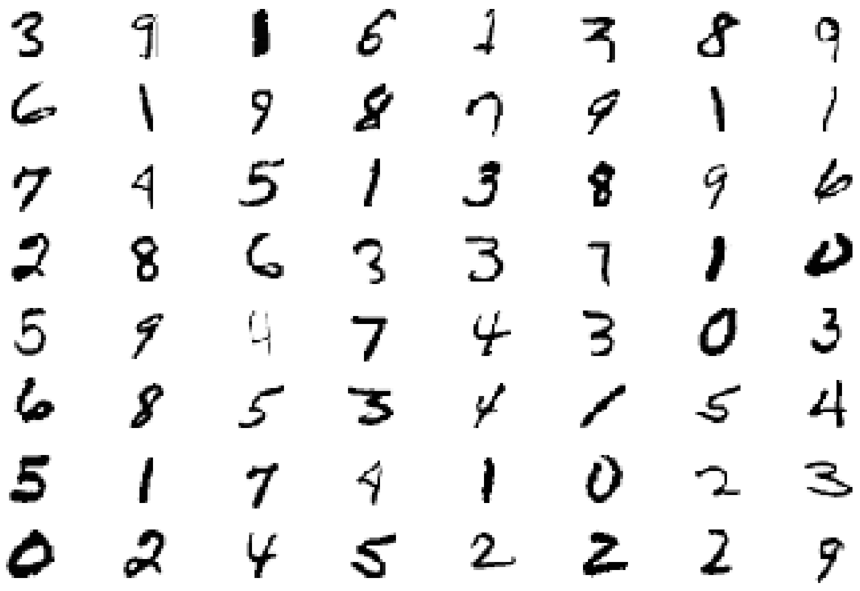
```
import numpy as np
import tensorflow as tf
import keras
from keras.datasets import mnist
from keras.layers import Dense
from keras.models import Sequential
from matplotlib import pyplot as plt
from random import randint

# Preparing the dataset
# Setup train and test splits
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Making a copy before flattening for the next code-segment which displays images
x_train_drawing = x_train
x_test_drawing = x_test

image_size = 784 # 28 x 28
x_train = x_train.reshape(x_train.shape[0], image_size)
x_test = x_test.reshape(x_test.shape[0], image_size)

# Convert class vectors to binary class matrices
num_classes = 10
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
```

```
/Users/kyl05002/opt/anaconda3/lib/python3.7/site-packages/tensorflow/python/framework/dtypes
  _np_qint8 = np.dtype([("qint8", np.int8, 1)])
/Users/kyl05002/opt/anaconda3/lib/python3.7/site-packages/tensorflow/python/framework/dtypes
  _np_quint8 = np.dtype([("quint8", np.uint8, 1)])
/Users/kyl05002/opt/anaconda3/lib/python3.7/site-packages/tensorflow/python/framework/dtypes
  _np_qint16 = np.dtype([("qint16", np.int16, 1)])
/Users/kyl05002/opt/anaconda3/lib/python3.7/site-packages/tensorflow/python/framework/dtypes
  _np_quint16 = np.dtype([("quint16", np.uint16, 1)])
/Users/kyl05002/opt/anaconda3/lib/python3.7/site-packages/tensorflow/python/framework/dtypes
  _np_qint32 = np.dtype([("qint32", np.int32, 1)])
/Users/kyl05002/opt/anaconda3/lib/python3.7/site-packages/tensorflow/python/framework/dtypes
  np_resource = np.dtype([("resource", np.ubyte, 1)])
/Users/kyl05002/opt/anaconda3/lib/python3.7/site-packages/tensorboard/compat/tensorflow_stub/
  _np_qint8 = np.dtype([("qint8", np.int8, 1)])
/Users/kyl05002/opt/anaconda3/lib/python3.7/site-packages/tensorboard/compat/tensorflow_stub/
  _np_quint8 = np.dtype([("quint8", np.uint8, 1)])
/Users/kyl05002/opt/anaconda3/lib/python3.7/site-packages/tensorboard/compat/tensorflow_stub/
  _np_qint16 = np.dtype([("qint16", np.int16, 1)])
/Users/kyl05002/opt/anaconda3/lib/python3.7/site-packages/tensorboard/compat/tensorflow_stub/
  _np_quint16 = np.dtype([("quint16", np.uint16, 1)])
/Users/kyl05002/opt/anaconda3/lib/python3.7/site-packages/tensorboard/compat/tensorflow_stub/
  _np_qint32 = np.dtype([("qint32", np.int32, 1)])
/Users/kyl05002/opt/anaconda3/lib/python3.7/site-packages/tensorboard/compat/tensorflow_stub/
  np_resource = np.dtype([("resource", np.ubyte, 1)])
Using TensorFlow backend.
```

J.1 A Look At Some Random Digits

Let us look at randomly selected digits from the training set.

```
for i in range(64):
    ax = plt.subplot(8, 8, i+1)
    ax.axis('off')
    plt.imshow(x_train_drawing[randint(0, x_train.shape[0])], cmap='Greys')
```



J.2 Neural Network

We design a neural network to solve MNIST. It has a single hidden layer with 32 nodes.

```
model = Sequential()

# The input layer requires the special input_shape parameter which should match
# the shape of our training data.
model.add(Dense(units=32, activation='sigmoid', input_shape=(image_size,)))
model.add(Dense(units=num_classes, activation='softmax'))
model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_3 (Dense)              (None, 32)                25120
_________________________________________________________________
dense_4 (Dense)              (None, 10)                330
=================================================================
Total params: 25,450
Trainable params: 25,450
```

```
Non-trainable params: 0
_________________________________________________________________
```

J.3 Train & Evaluate The Network

This code trains and evaluates the model we defined above.

```
model.compile(optimizer="sgd", loss='categorical_crossentropy', metrics=['accuracy'])
```

```
model.fit(x_train, y_train, batch_size=128, epochs=5)
```

```
WARNING:tensorflow:From /Users/kyl05002/opt/anaconda3/lib/python3.7/site-packages/keras/back

Epoch 1/5
60000/60000 [==============================] - 1s 17us/step - loss: 0.7042 - accuracy: 0.820
Epoch 2/5
60000/60000 [==============================] - 1s 15us/step - loss: 0.3572 - accuracy: 0.907
Epoch 3/5
60000/60000 [==============================] - 1s 15us/step - loss: 0.2916 - accuracy: 0.922
Epoch 4/5
60000/60000 [==============================] - 1s 15us/step - loss: 0.2547 - accuracy: 0.932
Epoch 5/5
60000/60000 [==============================] - 1s 15us/step - loss: 0.2291 - accuracy: 0.938

<keras.callbacks.callbacks.History at 0x7fc626ea7890>
```

```
loss, accuracy  = model.evaluate(x_test, y_test)
print(f'Test accuracy: {accuracy:.3}')
```

```
10000/10000 [==============================] - 0s 13us/step
Test accuracy: 0.888
```

```
x_test
```

```
array([[0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       ...,
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0]], dtype=uint8)
```

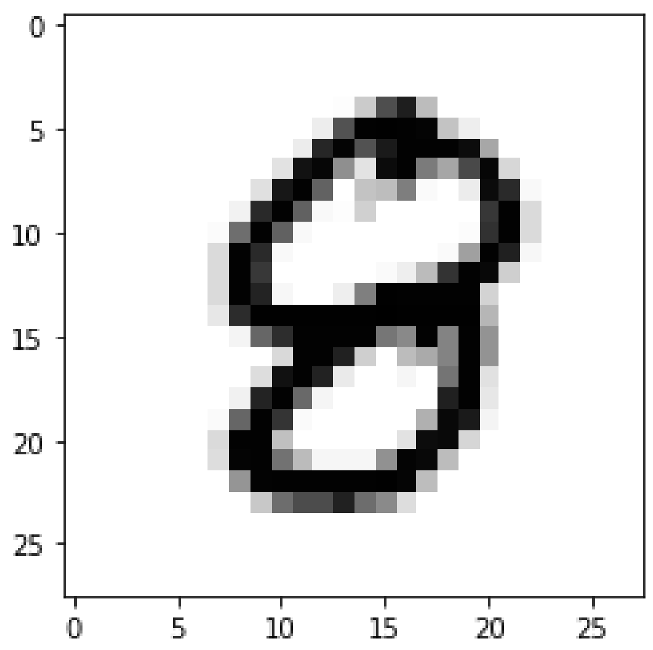
```
y_pred = model.predict(x_test)
```

```
print(y_pred[:9].round())
print(y_test[:9])
```

```
[[0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
[[0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]]
```

```
rd=randint(0, x_test.shape[0])
plt.imshow(x_test_drawing[rd], cmap='Greys')
```

```
<matplotlib.image.AxesImage at 0x7fc7292f1290>
```



```
print(y_pred[rd])
print(y_pred[rd].round())
```

```
[0.02834265 0.01066917 0.03439402 0.0220306  0.0111017  0.03483087
 0.02701708 0.0030446  0.7904302  0.0381391 ]
[0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
```