Aprendizagem Automática (Machine Learning)

Lecture Notes

Chapter 1

Introduction and course overview

What is machine learning and what can we use it for. Fundamental concepts. Different types of learning. Outline of the course.

Note: for details on assignments, class schedules and assessment, please refer to the course page

1.1 What is machine learning

Machine learning is the science of building systems that improve with data. This is a broad concept that includes instances ranging from self-driving cars to sorting images on a database and from recommendation systems for diagnosing diseases to fitting parameters in climate change models. The fundamental idea is that the system can use data to improve its performance at some task. Which immediately points us to the three basic elements of a well-posed machine learning problem:

1. The task that the system must perform.
2. The measure by which its performance can be evaluated.
3. The data that can be used to improve its performance.

For example, suppose we want to automate airline ticket purchasing by phone. The task to perform is thus to identify the requests, such as asking to book a flight, the origin and destination, the required flight and so on. The system's performance at this task can be measured by the frequency of correctly identified expressions. In the work reported in [9], the data the used was a corpus of manually annotated expressions. This is an example of the system parsing spoken sentences and identifying the relevant elements for processing the requests:

```
<book_flight> please book me on </book_flight> <numflt> flight twenty one </numflt>

<i_want_to_go> i would like to fly </i_want_to_go>
<city_from> from philadelphia </city_from> <city_to> to dallas </city_to>

<request1> could you please list the </request1> flights
<city_from> from boston </city_from> <city_to> to denver </city_to>
on <date> july twenty eighth </date>
```

Different tasks will determine different approaches. We may want to predict some continuous value, such as the price of apartments, which is a *Regression* problem. Or we may have a *Classification*, when we want to predict in which category, from a discrete set, each example belongs to. If we do this from a set of data containing the right answers, so we can then extrapolate to new examples, we are doing *Supervised Learning*.

We may want to find *Association Rules*, which are joint or conditional probability distributions of some features. For example, which products customers tend to purchase together, so we can optimize their placement in supermarket aisles. We may want to do *Density Estimation* to understand how feature values are distributed, or perhaps group examples according to some similarity measure, which is called *Clustering*. For example, grouping images together according to how similar they look. These are examples of *Unsupervised Learning*, because in these cases there is no *Ground Truth* in the data that can tell us if we made the right or wrong choice.

Supervised Learning requires that all data be labelled and *Unsupervised Learning* uses only unlabelled data. But it is possible to use data sets in which some data is labelled but the rest, usually most of the data, is not. In this case, we have *Semi-supervised Learning*. This approach has the advantage that, usually, unlabelled data is much easier to find than correctly labelled data. For example, it is possible to obtain from the World Wide Web many examples of English texts but to label correctly each grammatical element of each sentence would be very laborious. By combining clustering and classification it is possible to use unlabelled texts to improve the parsing and classification of elements from a set of labelled texts.

Some tasks involve sequences of decisions, such as when playing a game, and the outcome can only be assessed after all decisions are made (e.g. win or lose). This is an example of a *Reinforcement Learning* problem, where each move is not good or bad by itself but only in the context of the sequence of moves.

With such a diverse range of applications and problems, Machine Learning benefits from contributions of many other disciplines. Computer Science, evidently, from subjects such as artificial intelligence, algorithms, complexity analysis and data management. Statistics is also important for inference, experiment design and data analysis. Mathematics is also crucial, with numerical methods at the base of most machine learning algorithms and probability theory underlying many machine learning approaches. Finally, Machine Learning is strongly inspired by Neuroscience, in particular perception, learning and memory, and Philosophy, especially epistemology and ontology.

1.2 Why machine learning is useful

Machine learning is useful if we cannot, or do not wish to, explicitly program a solution to our problems. For example, humans can easily identify handwritten digits such as those in zip codes of mail addresses. However, it is not easy to find specific rules for programming a computer to automate this task. This is a classic example of a problem where machine learning is a useful solution. For decades now, machine learning algorithms have been applied to automating identification of handwritten digits [7]. Figure 1.1 shows an example of this problem.

Machine learning is also becoming increasingly more useful as the amount, complexity and quality of data increases. Recently, the trend has been towards an exponential growth in data.

Another reason for using machine learning is so that the system can adapt to changing conditions. If we have a static set of data we may figure out some rules for organizing and grouping the examples after careful examination of the data. However, if the data set is continuously changing, as is generally

Figure 1.1: Handwritten zip code digits

the case for most applications, it is not feasible to have programmers dedicated to constantly adapting the code to extract information from the data. In these cases, automated systems that can constantly learn from the new data are crucial. An example of this is the optimization of search engines. The search engine must interpret the user's query, consider how to expand the search by using synonyms or words with overlapping meaning, and, especially, how to rank the results. These systems constantly learn from the users, remembering which links are preferred, which search terms are most used and their associations, and a large amount of information on the user (often even arguably violating privacy rights).

Machine learning raises some important technical challenges, and even some ethical issues regarding the information that is used and the purpose for which it is used, but it is clear that machine learning is an important field in computer science and its importance can only grow as data and computation power keep growing.

1.3 Fundamental concepts

Throughout this course we will constantly rely on some important concepts and it is important that they are clear from the beginning. First is the concept of the *hypothesis class*. This is the space of possibilities in which we will try to optimize the solution to our machine learning problem. Suppose we have the data set represented in Figure 1.2, where each point has two continuous features, represented in the X and Y axes, and is labelled either in the red or blue class.

One possible way of separating them would be to try to find the horizontal line that best splits the two classes. In this case, our *hypothesis class* would be the set of horizontal lines, as represented in Figure 1.3.

Machine learning is closely associated with statistics and so the term *model* is also used to refer to a representation of a *hypothesis class*, typically using some parameters. For example, we could describe this set of all horizontal lines with the parametric model $y = \theta$, where θ is the parameter to adjust in order to instantiate the model into a specific line. This is an hypothesis in the *hypothesis class*. In the literature, and in this course, it is common to find both *model* and *hypothesis class* to refer to the set of possible instances in which we want to find the best solution to our learning problem. An alternative to the horizontal line model would be to consider all circles of radius 1 and try to find which of these circles includes all blue points and excludes red points (Figure 1.4).

This different *hypothesis class* allow us to find different *hypotheses* that cannot be expressed with the $y = \theta$ model. In this case, we would have a model with two parameters, $(x - \theta_1)^2 + (y - \theta_2)^2 = 1$.

We can say that the circle of radius 1 centered at $(-1, -1)$ is an instance of the $(x-\theta_1)^2+(y-\theta_2)^2 = 1$ model, or a hypothesis from this hypothesis class, and the line $y = 0$ is an instance of the $y = \theta$

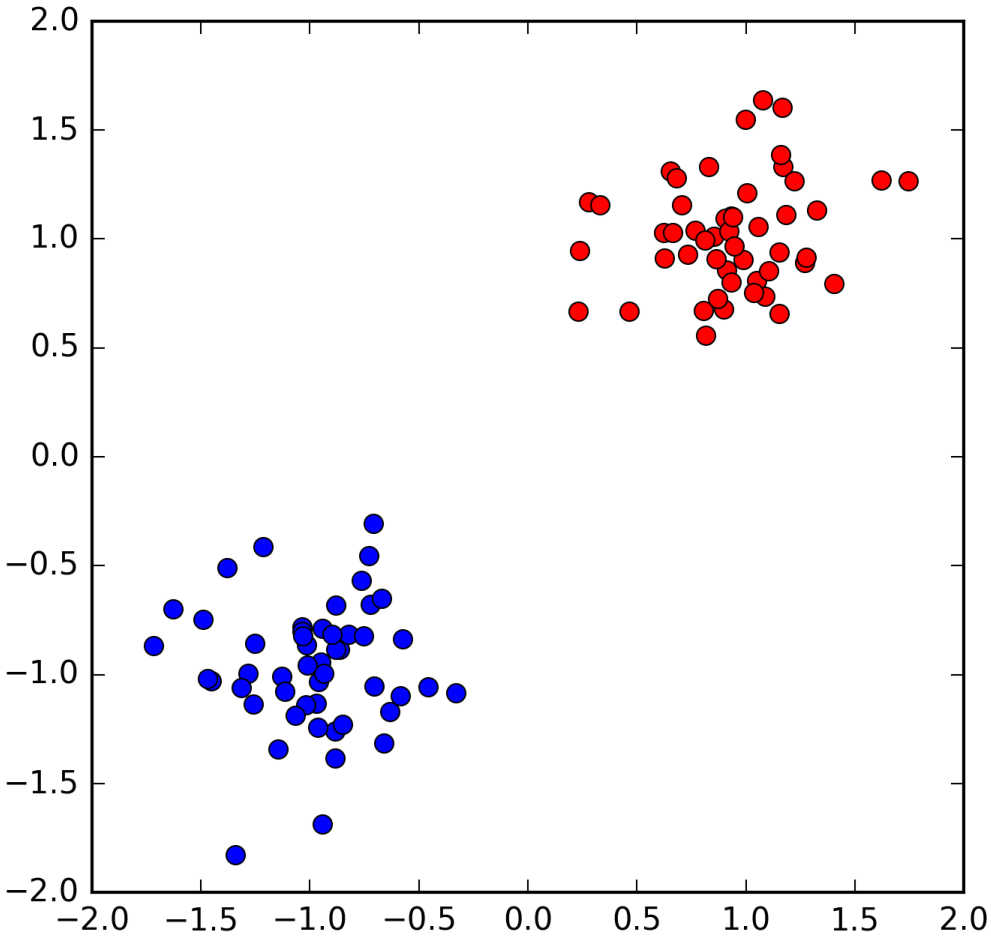

Figure 1.2: Arbitrary data set in two dimensions, divided into two classes (red and blue)

model or an hypothesis from that hypothesis class. The important point here is to distinguish between the universe of possibilities we are considering, which is the *model* or *hypothesis class*, and the specific instantiation of the model, or *hypothesis*, we obtain by setting the parameters and which constitutes an answer to the learning problem.

The *hypothesis class* determines the *inductive bias* of our learning system. We cannot learn anything if we do not assume anything, because this would make it impossible to extrapolate from the data we have, and so we must make some prior assumptions about the problem we are solving. For example, assuming that we can separate the red and blue classes using some horizontal line or some circle of radius 1. This is such a crucial point that we will often talk about the problem of *model selection*, which consists of finding the best *hypothesis class* for a particular problem.

1.4 Overview of Machine Learning problems

As mentioned above, one class of machine learning problems is *Unsupervised learning*, involving unlabelled examples. In this case, the objective is generally to obtain some information about the structure of the data. A schematic representation of the unsupervised learning process is shown in Figure 1.5

For example, in *clustering*, we may want to organize the data so as to group together similar data points. An example of this is clustering of images obtained from the World Wide Web, to help guide

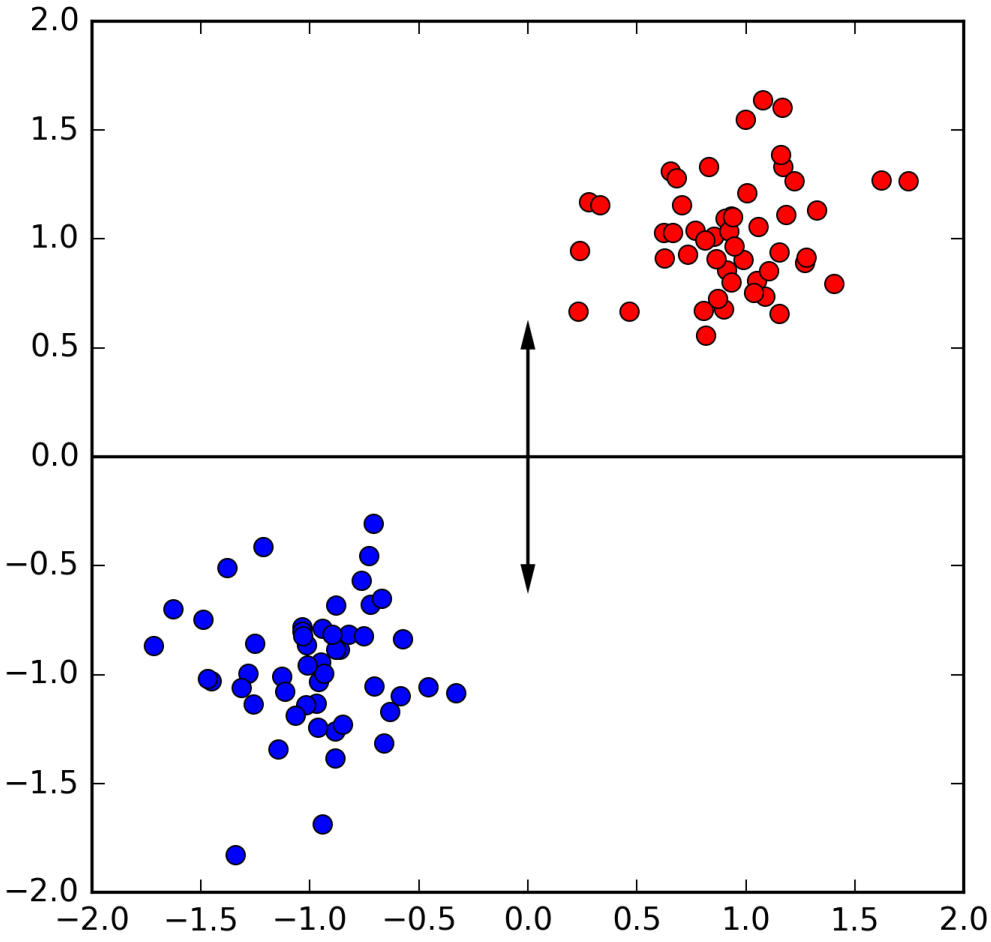

Figure 1.3: One hytpothesis class: horizontal lines to divide the two classes

organize search results. The authors of [5] extracted features both from the images themselves as well as from the text of the pages where the images were found and hypelinks between them. Figure 1.6 gives an example of the resulting clusters for an image search with the keyword "pluto".

Unsupervised learning gives us new values we can associate with the original data, and so *unsupervised learning* can be used as part of a larger learning task. This is what happens in deep learning, for example.

In *Supervised Learning*, we have a fully labelled data set that provides us not only with the input features for our learning machine but also with the correct answers, allowing us to supervise the learning process directly. Schematically, supervised learning looks like the diagram in Figure 1.7:

From the data we feed into the learner those features that will be used in the future to predict something about new data. But we also use the target values to compute the *empirical error* of the learner during the learning process. In this way, we can improve its performance in correctly predicting the target values.

An example of this is given in [20]. The task consists of identifying faces in photographs. It is a *Classification* task because each segment of the image may be classified as either a face or not a face. The data used for training the classifier consists of a set of labelled images of faces and a set of labelled images that were not faces. Figure 1.8 shows, on the right, an example of the set of face images used in training (non-face images are not shown here) and an example of the application of the final classifier.

The authors of [20] also report a *semi-supervised learning* approach, where the labelled examples

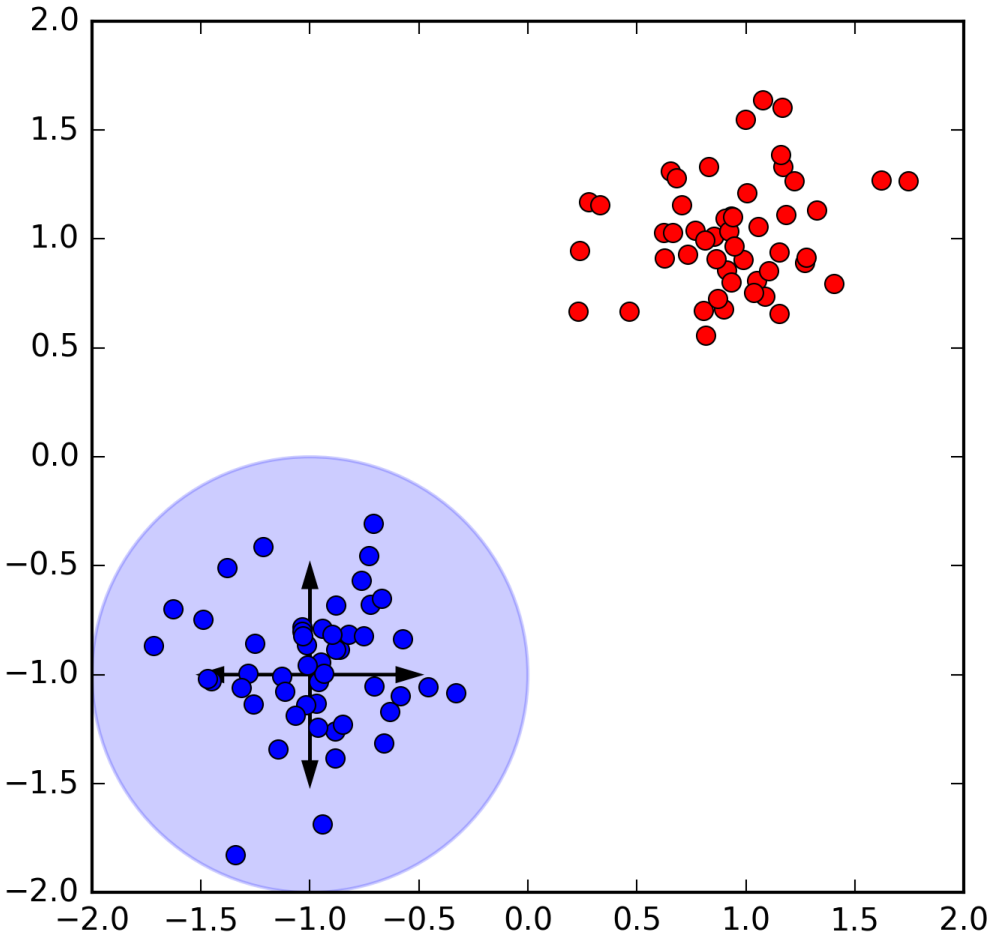

Figure 1.4: Another hytpothesis class: circles of radius 1 to isolate one class from the other

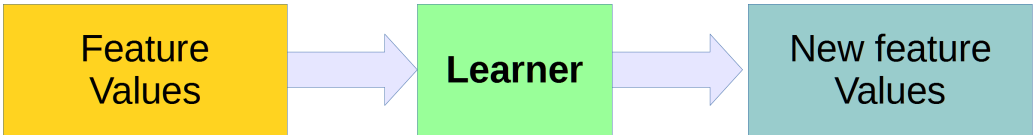

Figure 1.5: Diagram of an unsupervised learning process

used to train the classifier are enriched with unlabelled image data. This requires accounting both for the statistical structure of all data, including the unlabelled data, and the classifier's performance in the labelled data. Another type of machine learning problem is *Reinforcement Learning*. In this type of problem, the task is to optimize some output, like game moves, for example, but the feedback guiding the learner must be given by some heuristic or an evaluation of an eventual outcome and not given by the data. So the learner must improve performance by improving the feedback (e.g. win or lose the game). Figure 1.9 shows a diagram of this learning process.

Some examples of reinforcement learning applications include robotics, for locomotion control and other tasks such as object manipulation, autonomous vehicle control, operations research (pricing, routing, marketing), and games. In this course we will focus on supervised and unsupervised learning and will not cover semi-supervised or reinforcement learning problems.

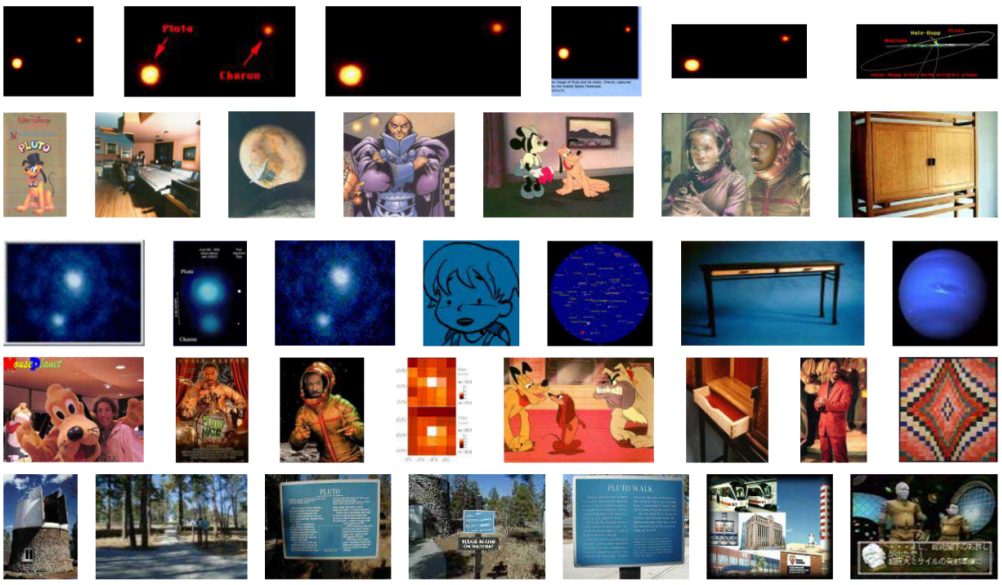

Figure 1.6: Clusters obtained for the image results of a search for “pluto”. Each cluster is a row of images. Figure from [5].

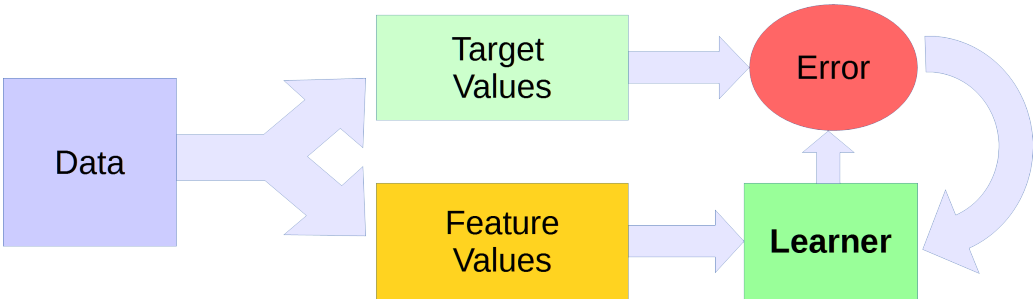

Figure 1.7: Diagram of a supervised learning process

Figure 1.8: Labelled face images used in training (left) and the result of applying the classifier (right). For more details see the original paper [20].

1.5 Goals and course outline

The main goal of this course is to provide a foundation on theoretical and practical aspects of machine learning so the student can get some experience with common machine learning techniques, understand the concepts, be able to follow the literature, acquire the skills to handle scientific computation problems and understand the algorithms from their mathematical specifications.

The first part of this course will focus on supervised learning. Broadly speaking, the task of learning

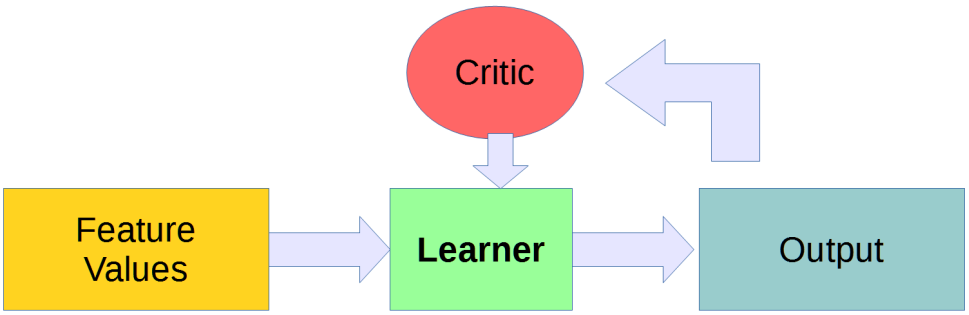

Figure 1.9: Diagram of a reinforcement learning process

how to predict an attribute of a universe of examples from a data set which includes both the observed features used for the prediction and the attribute to be predicted. The second part will cover some basics of learning theory and more detailed aspects of model selection. The third part will be dedicated to ensemble methods and the final part to unsupervised learning algorithms.

1.6 Further Reading

1. Alpaydin [2], Chapter 1
2. Mitchell [18], Chapter 1
3. Marsland [17], Chapter 1, sections 1.1 through 1.4.

Part I

Supervised Learning

Chapter 2

Introduction to supervised learning

Basic concepts of supervised learning. Empirical error. Maximum likelihood and error minimization. A regression example: curve fitting by least mean squares minimization. Curve fitting as a linear regression.

2.1 Supervised learning

We call *Supervised Learning* the task of learning to predict attributes from data that include those attributes. More formally, we have a set of examples with features X and some label Y, $\{(x^1, y^1), ..., (x^n, y^n)\}$, and we assume there is some unknown function $F(X) : X \rightarrow Y$. Our goal is to find a function $g(\theta, X) : X \rightarrow Y$, which is a function of some set of parameters θ, that approximates the unknown $F(X) : X \rightarrow Y$ and can tells us the Y values of any examples, even if not from our known set.

The reason for calling this supervised learning is that, by having all the Y values, we can supervise the learning process by comparing the predicted values to the known values in the data. This allows us to empirically estimate the error of each hypothesis.

The *empirical error*, or *training error*, is a measure of how any hypothesis obtained by instantiating the parameters θ of our model (the $g(\theta, X)$ set of functions in our *hypothesis class*) performs in predicting the Y values of the training data. Thus, we can formulate one machine learning problem in this way:

1. The task: Predict the Y values in the $\{(x_1, y_1), ..., (x_n, y_n)\}$ set.
2. The performance measure: training error, using Y.
3. The data: the $\{(x_1, y_1), ..., (x_n, y_n)\}$ set.

Note that this is not a very useful problem to solve because this only aims at predicting the values that we already know. In other words, this tries to approximate the unknown function $F(X) : X \rightarrow Y$ only within our known data set. A better alternative would be to find the hypothesis that would minimize the error for any examples, even those not included in the training set. That is usually the goal of a machine learning application. But we'll set aside that complication for now and focus on the simplified problem first.

Supervised learning problems can be split in two different categories. *Classification* problems are those in which the Y values belong to discrete categories. For example, the classification of email messages into spam and not spam or the classification of mushrooms into edible and poisonous categories. In this case the error can be something like the percentage of misclassified examples. *Regression* problems are those in which the Y values are continuous. We will focus on *regression* in this chapter and the next, and cover *classification* later on.

2.2 Linear Regression

A *linear regression* is a regression in which the hypothesis class corresponds to the model $y = \theta_1 x_1 + \theta_2 x_2 + ... + \theta_{n+1}$, where each x_n is one dimension of the input space. Let's suppose, to simplify, that our input space has only one dimension and we have a set of (x, y) points and want to find the best way to predict the y value of each point given the x value assuming some specific *hypothesis class*. Let us suppose the *hypothesis class* is the set of all straight lines, defined by the parameters of the model $y = \theta_1 x + \theta_2$. Figure 2.1 shows an example of a data set of points and possible lines from our *hypothesis class*, obtained by instantiating the model with different values of θ_1 and θ_2.

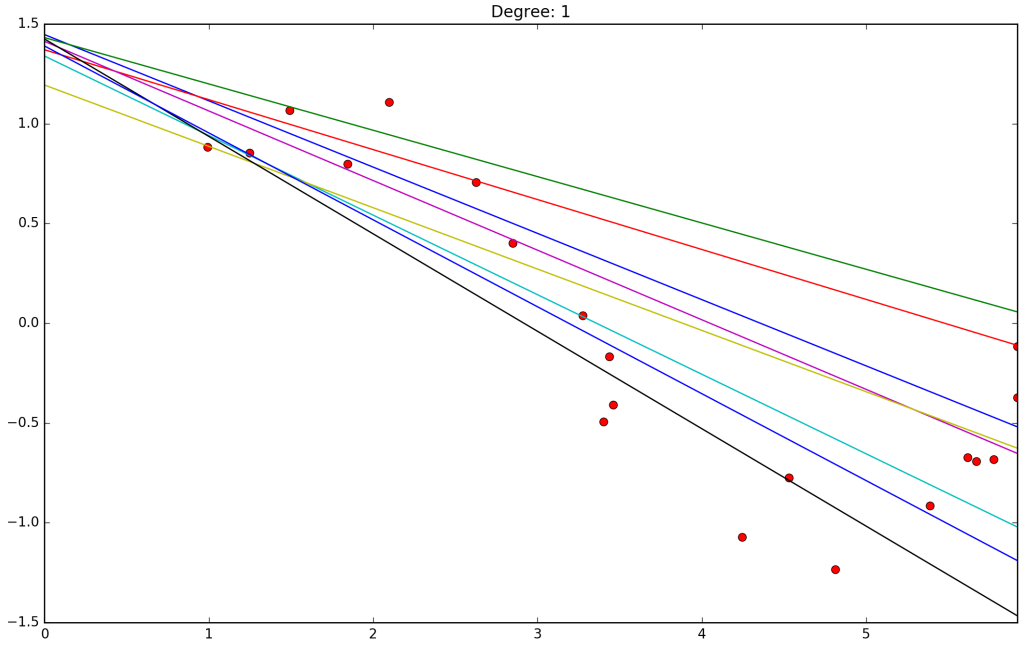

Figure 2.1: Example of lines for predicting the y values in these data.

How can we determine the best line? Let us assume that the dependent variable y is some (unknown) function of the independent variable x plus some error:

$$y = F(x) + \epsilon$$

We want to approximate $F(x)$ with a model $g(x, \theta_1, \theta_2)$. Assuming that the error is random and normally distributed:

$$\epsilon \sim N(0, \sigma^2)$$

then, if $g(x, \theta_1, \theta_2)$ is a good approximation of the true function $F(x)$, the probability of having a particular y value given some x value can be computed from our function $g(x, \theta_1, \theta_2)$ as:

$$p(y|x) \sim \mathcal{N}(g(x, \theta_1, \theta_2), \sigma^2)$$

This allows us to estimate the probability of the data coming out with the distribution we observe in our data set given any hypothesis instantiating θ, representing the vector of all $\theta_1, ..., \theta_n$ parameters (in this case, θ_1, θ_2). The probability of the data given the hypothesis is the *likelihood* of the hypothesis. Note that we cannot assume a probability for the hypothesis, at least in a frequentist sense, because the hypothesis is not a random variable. What we assume to be random here is the sampling of data that resulted in obtaining this dataset from the universe of all possible data.

Thus, given our dataset $\mathcal{X} = \{x^t, y^t\}_{t=1}^N$ and knowing that $p(x, y) = p(y|y)p(x)$, then the likelihood of the set of parameters θ is

$$l(\theta|\mathcal{X}) = \prod_{t=1}^{n} p(x^t, y^t) = \prod_{t=1}^{n} p(y^t|x^t) \times \prod_{t=1}^{n} p(x^t)$$

Now we know how to choose the best hypothesis: we pick the one with the *maximum likelihood*. In other words, we pick the hypothesis that estimates the largest probability of obtaining the data we have. This is a generic approach that is often used in machine learning. But, to simplify the math, let us change the expression. First, we know that the hypothesis that maximizes the likelihood also maximizes the logarithm of the likelihood, so we can focus on the logarithm of the likelihood, $\mathcal{L}$, instead of the likelihood l:

$$\mathcal{L}(\theta|\mathcal{X}) = log \prod_{t=1}^{n} p(y^t|x^t) + log \prod_{t=1}^{n} p(x^t)$$

We can also ignore the $p(x)$ term since this corresponds to the probability of drawing those x values in our data from the universe of possible values and this is the same for all hypotheses (all values of θ) we are considering.

$$\mathcal{L}(\theta|\mathcal{X}) \propto log \prod_{t=1}^{n} p(y^t|x^t)$$

Since we assume that the probability of obtaining some y value given some x is approximately normally distributed around our prediction, we can replace that term with the corresponding distribution:

$$p(y|x) \sim \mathcal{N}(g(x, \theta), \sigma^2)$$

and then replace it with the expression for the normal distribution:

$$\mathcal{N}(z, \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(z-\mu)^2/2\sigma^2}$$

leaving:

$$\mathcal{L}(\theta|\mathcal{X}) \propto log \prod_{t=1}^{n} \frac{1}{\sigma\sqrt{2\pi}} e^{-[y^t - g(x^t|\theta)]^2/2\sigma^2}$$

which can be simplified to:

$$\mathcal{L}(\theta|\mathcal{X}) \propto log \prod_{t=1}^{n} e^{-[y^t - g(x^t|\theta)]^2}$$

$$\mathcal{L}(\theta|\mathcal{X}) \propto -\sum_{t=1}^{n} [y^t - g(x^t|\theta)]^2$$

But this is the expression of the square of the training error:

$$E(\theta|\mathcal{X}) = \sum_{t=1}^{n} [y^t - g(x^t|\theta)]^2$$

So, basically, to find the hypothesis with the *maximum likelihood* we need (under our assumptions) to find the hypothesis with the *minimum squared error* on our training set. This problem is called a *Least Mean Squares minimization.*

Note that the squared error is often represented by this expression:

$$E(\theta|\mathcal{X}) = \frac{1}{2}\sum_{t=1}^{n}[y^t - g(x^t|\theta)]^2$$

The reason for this is that, when computing the derivative of this error as a function of the parameters, the square power cancels the 2 in the denominator, simplifying the algebra. However, the values obtained for the parameters minimizing the squared error or one half the squared error are the same. This is merely an algebraic convenience.

2.3 Least Mean Squares minimization

In our straight line model (Figure 2.1) we need to consider two parameters, θ_1 and θ_2. If we compute the squared error for all combinations of parameters in some range we will obtain something like shown in Figure 2.2.

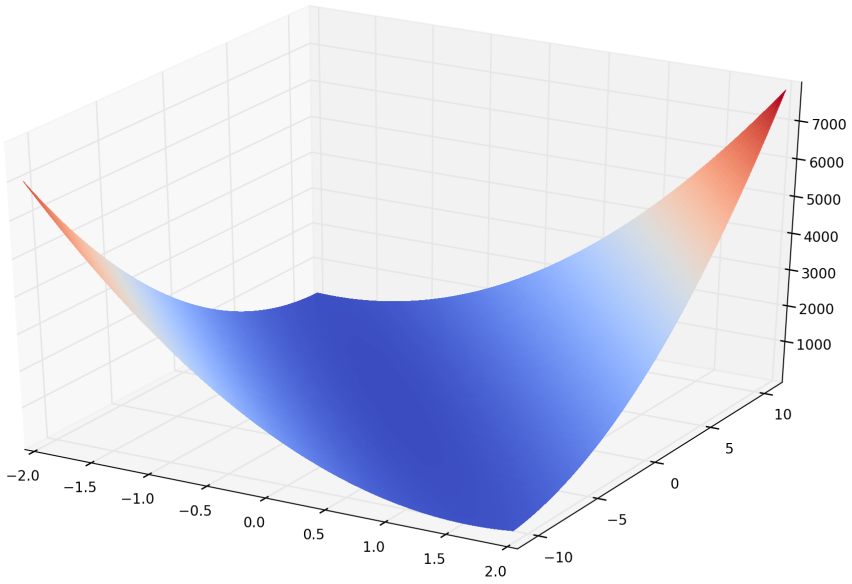

Figure 2.2: Surface of the squared error function for two parameters.

To find the optimal combination of parameters we need to find the lowest point of this surface. Thus we use a gradient descent algorithm that starts from an arbitrary point as an initial guess and then proceeds to descend the error surface in different directions until converging to the desired minimum. This is illustraded in Figure 2.3

This will be a useful approach in many different problems we will encounter. The important idea is this: if we assume that our model approaches the desired target values with some normal error, as a function of the features in our dataset and the parameters of the model, then maximizing the likelihood of our parameters is equivalent to minimizing the squared error. We shall see in the next chapter that considering only the training error may not be a good idea, but in any case this *least mean squares minimization* approach is an important tool in machine learning.

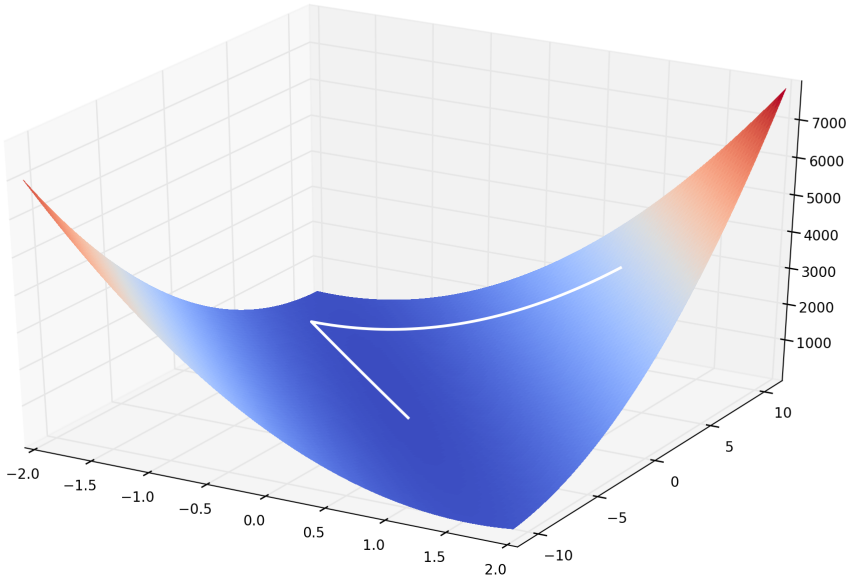

Figure 2.3: Gradient descent on the squared error surface.

In this way, we can find the hypothesis that best fits the data. The straight line that minimizes the squared error for our data set is shown in Figure 2.4. One thing we can notice immediately is that, despite being the best straight line to predict the y values in our data, it is still a very poor predictor of these values. We need to change your *hypothesis class* and try to find different hypotheses.

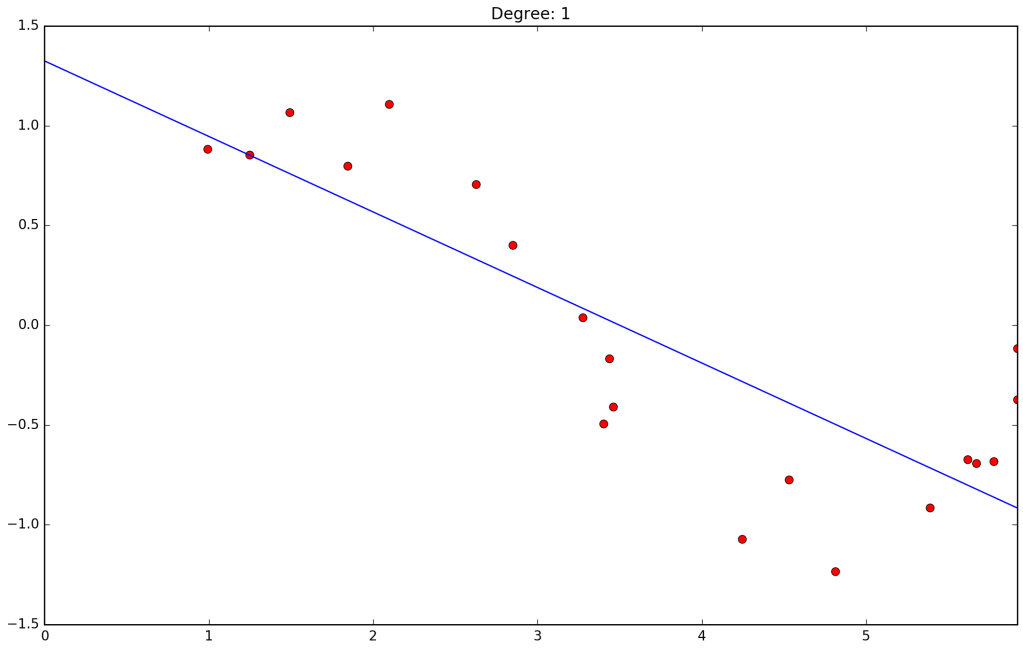

Figure 2.4: Best straight line for predicting the y values in our data.

2.4 Beyond the straight

Since fitting a straight line to these data is so evidently inadequate, we can try to consider alternatives. Let us start by changing the data. We have a set of values for x and y, and we fit a straight line that gives

us y given each x. But imagine that we spread our points over a plane with coordinates (x_1, x_2) instead of x, and found the plane that minimized the error between to the y values in this new space. We are still fitting a "straight" function, it's just in more dimensions than the initial one. So let us compute this new data set $\mathcal{X}_{\in} = \{x_1^t, x_2^t, y^t\}$ by making $x_1 = x^2$ and $x_2 = x$ for each point in the original set. This gives us the data set represented in Figure 2.5

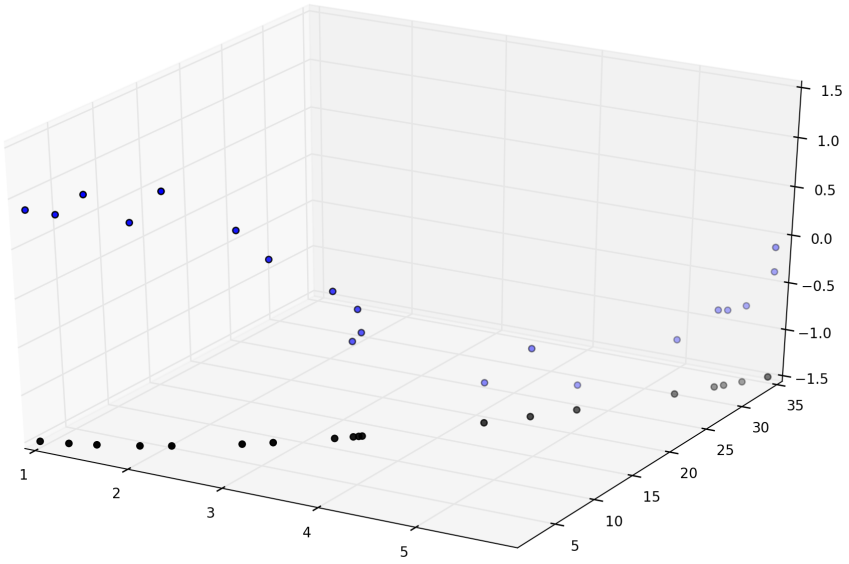

Figure 2.5: Transformed data set. Note that the black dots below are just the "shadows" to indicate the projection of the data in the (x^2, x) plane.

Now our model is the equation for the plane, which we can write as $y = \theta_1 x_1 + \theta_2 x_2 + \theta_3$, and each hypothesis in this hypothesis class will be a particular plane obtained by instantiating the three parameters. Minimizing the square error, we obtain the plane shown in Figure 2.6.

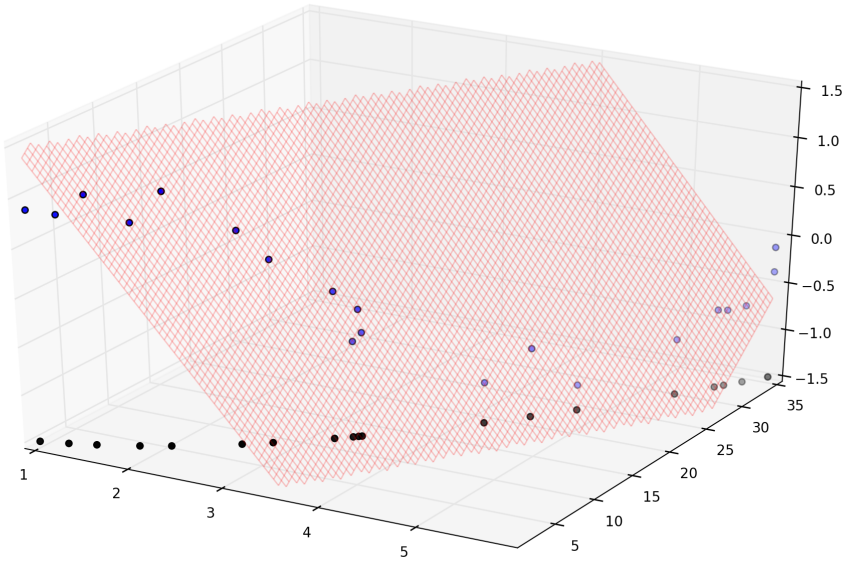

Figure 2.6: The plane that best fits the transformed data.

Now we can convert back into our original problem. We know that our data always falls in the line where $x_1 = x_2^2$, because that was our initial transformation. If we intersect the best plane we found with this line, we get a line that we can project into the initial (x, y) space. This line and the resulting projection is shown in Figure 2.7.

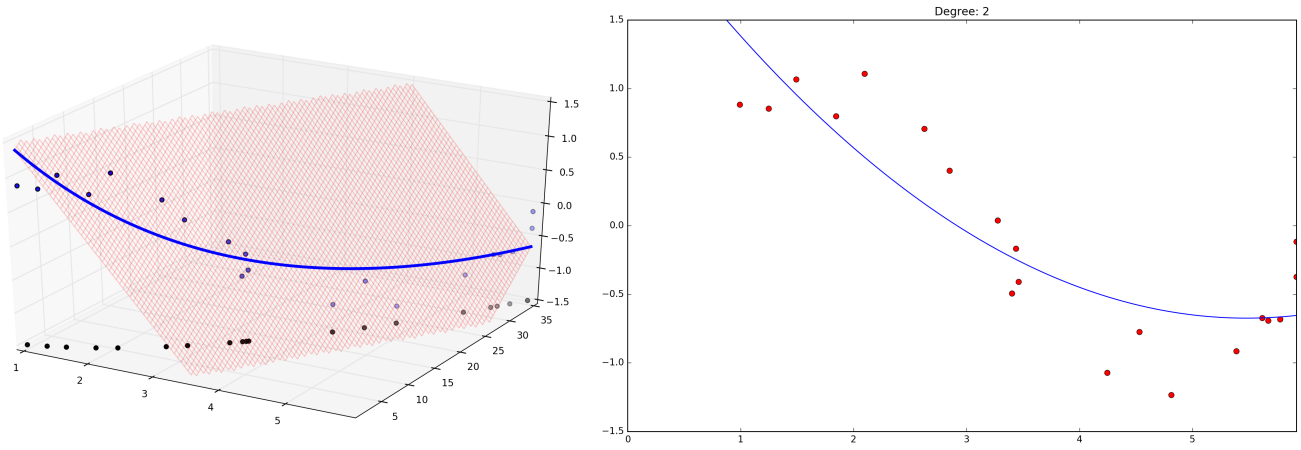

Figure 2.7: The line for the best fit projected back into the original data set.

This is the equivalent of doing a second degree polynomial regression on the original data. We could just have kept the original data set and just changed our model from the initial straight line, which is a first degree polynomial ($y = \theta_1 x + \theta_2$) to that of a quadratic curve, $y = \theta_1 x^2 + \theta_2 x + \theta_3$. In fact, this is the easiest way to solve this problem with the tools we are using in this course. Here is the code for loading the data and computing the best second degree polynomial.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 mat = np.loadtxt('polydata.csv',delimiter=';')
5 x,y = (mat[:,0], mat[:,1])
6 coefs = np.polyfit(x,y,2)
```

The first two lines import the `numpy` library for the computations and the `matplotlib` library for the plot, shown below. Then we load the `csv` file with the data, splitting the matrix into two variables `x` and `y`. Then line 6 does the actual work of computing the coefficients of the second degree polynomial. Now we can plot the results by first computing the polynomial over 100 points and plotting the data, the line and saving the figure. The code continues below.

```
7 pxs = np.linspace(0,max(x),100)
8 poly = np.polyval(coefs,pxs)
9
10 plt.figure(1, figsize=(12, 8), frameon=False)
11 plt.plot(x,y,'.r')
12 plt.plot(pxs,poly,'-')
13 plt.axis([0,max(x),-1.5,1.5])
14 plt.title('Degree: 2')
15 plt.savefig('testplot.png')
16 plt.close()
```

However, there is an important lesson here that will reveal its usefulness when we deal with more complex problems. We can use a simple hypothesis class, for example all linear relations of variables,

corresponding to all hyperplanes in an N-dimensional problem, and use that hypothesis class for fitting or classifying our data in complex ways by transforming our data into higher dimensional representations of the same problem. In this example, we saw how the (straight) plane we used in 3 dimensions to fit our data projects back into a curved line in the original problem with only one independent and one dependent dimensions. This is an important way of thinking about machine learning problems.

2.5 Getting carried away

If a quadratic curve fits our data better than a straight line, a third degree polynomial is even better. Or higher degrees. Figure 2.8 shows the result of fitting a third degree and a fifteenth degree polynomial to our data. The polynomial of degree 15 certainly fits the data better, greatly reducing the training error. But is this really the best option? We will discuss this problem in the next chapter and lecture.

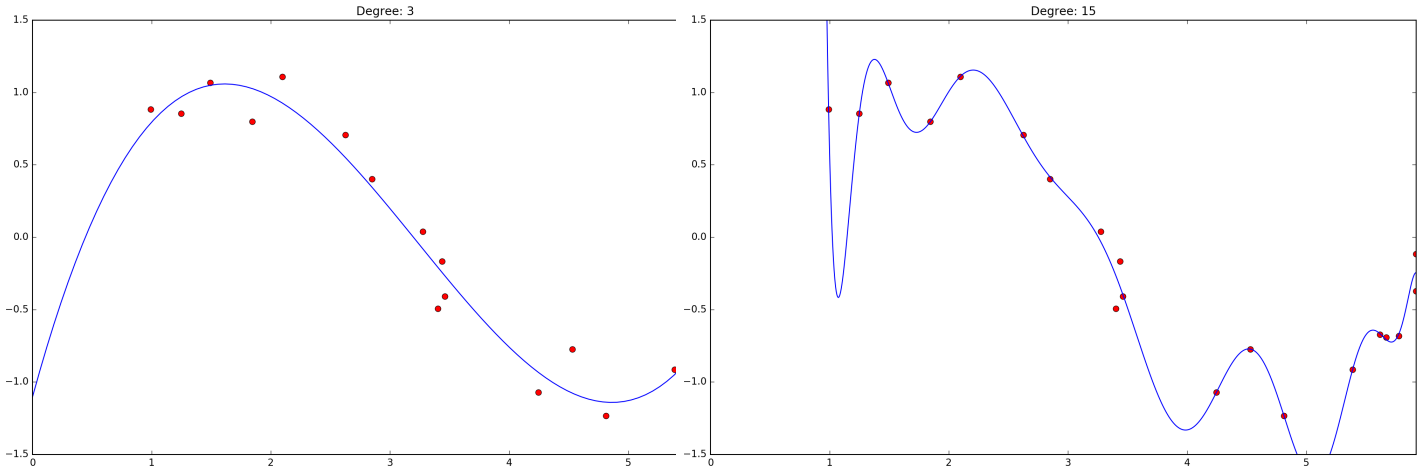

Figure 2.8: Fitting our data with polynomials of degree 3 and 15.

2.6 Summary

In this chapter we met several important ideas that we will revisit often during this course. First, we had to choose a *hypothesis class* for approximating the unknown function that determines the relations between the variables we are studying. Second, had to choose some measure of adjustment to select our parameters. In this case, we chose the *maximum likelihood*, which reduced to the *least squared error* measure for fitting our lines. Third, to adjust our parameters we needed to solve an optimization problem to find the values that optimize our adjustment measure. In this case, that minimize the squared error. Then we saw two more important ideas. One is that we can increase the power of linear models by increasing the number of features using non-linear transformations from the original feature values. The other is that this can result in fitting the known data too well. In the next chapter we will see how to address this last problem.

2.7 Further Reading

1. Bishop [4], Chapter 1
2. Alpaydin [2], Section 2.6

3. Marsland [17], Sections 1.4 and 2.4.

Chapter 3

Overfitting in Linear Regression

Overfitting. Training, validation and testing.

3.1 Estimating the true error

We saw in the previous lecture that we could improve the fit of a curve to a set of points by increasing the number of parameters. Figure 2.8 showed the difference between fitting the points with a third degree polynomial and a polynomial of degree 15. However, it is apparent that the result is fitting the known points by sacrificing the ability to correctly predict values not in the training set. This is a common concern in supervised learning problems.

In general, in a regression problem, we want to find a function to predict the Y values of elements of some universe $\mathcal{U}$ from their observable features X. The data is a labelled subset of $\mathcal{U}$, $\{(x_1, y_1), ..., (x_n, y_n)\}$, from which we can try to infer a function to predict Y and on which we can compute the *training error*, as we saw in the last lecture. However, the error we would like to minimise is the expected error over any element of $\mathcal{U}$, which is the *true error*, and not only the error measured in our training set (the *training error*).

One way to estimate the expected error in predicting the y value of any element of $\mathcal{U}$ is to reserve some elements of our data set specifically for this error estimate. These elements will not be used in training. Thus, we split our data into two sets: the *training set* and the *test set*. We use the *training set* to fit our function, minimizing the *training error*, and then the *test set* to estimate how our function performs in predicting the values of examples outside the *training set*. Figure 3.1 shows an example of fitting a fifth degree polynomial to 35 points in the data set (the training set, in blue) and then computing the *test error* with the remaining 15 points (shown in red).

The difference between the *test error* and the *training error* is called the *generalization error*, and is a measure of how well the learner can generalize from the training set to new examples.

It is important to note that the data set is split randomly between training and test sets. Depending on this selection, different curves will result from fitting the training set, with different test error values. Figure /ref3-diffsplits illustrates the result of a fifth degree polynomial fit to different subsets of the same data set, using the remaining points to evaluate the test error.

The *test error* is an unbiased estimate of the *true error*, but it is randomly distributed around the *true error*. This is because the *true error* is the average error for all the infinite possible data points while the *test error* is the error measured on the sample of points we assigned to the test set. The *test*

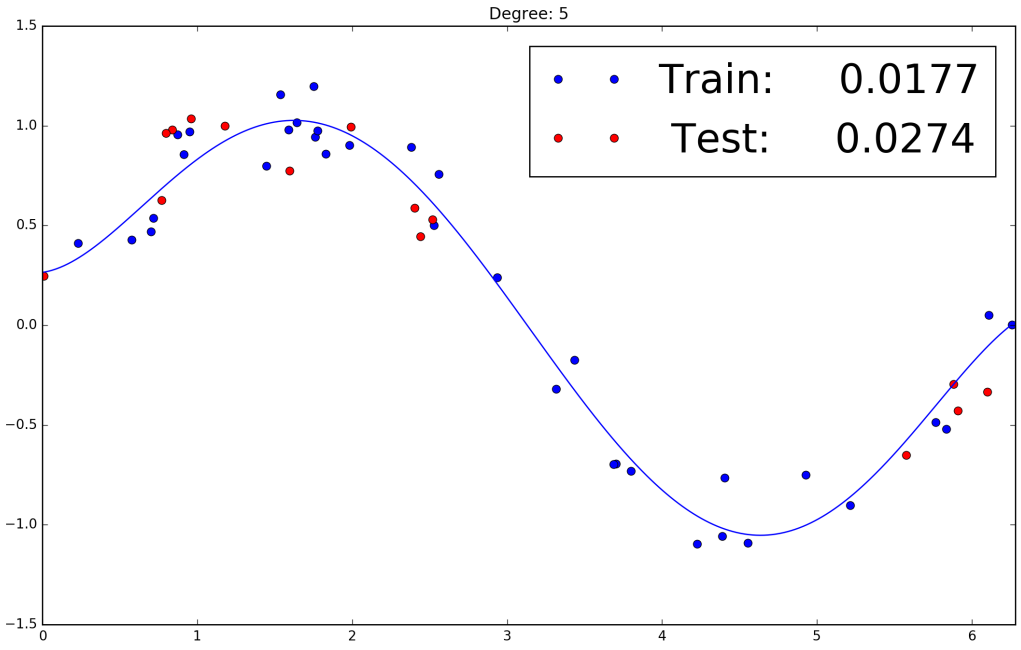

Figure 3.1: Training and test error example.

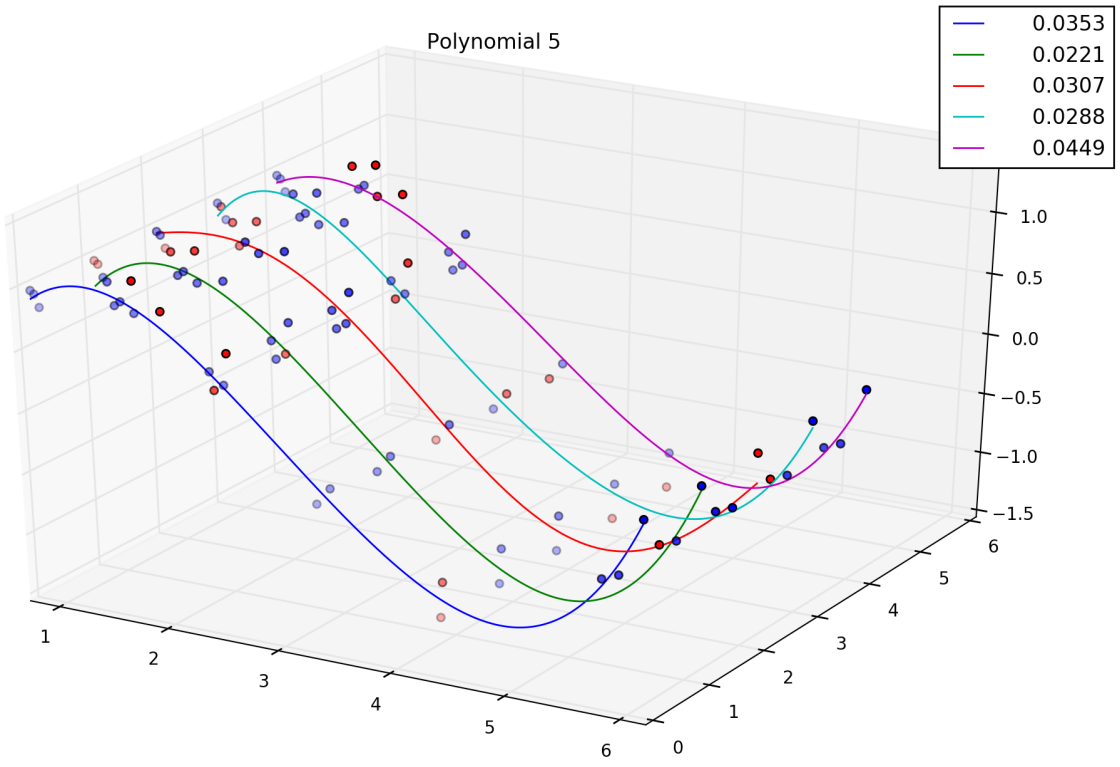

Figure 3.2: Fitting the same model to different training sets. The test error is indicated in the legend.

error is an unbiased estimator for the *true error* because, assuming the data set is a random sample of $\mathcal{U}$ and the training and test sets are randomly generated, then the average of the measured *test error*, over a large number of repetitions of the experiment, would tend towards the *true error*. However, a single measure of the *test error* will not correspond exactly to the *true error*. It is actually a sample from a probability distribution around the *true error*, like illustrated in Figure 3.3, because it depends on the points that were randomly assigned to the test set.

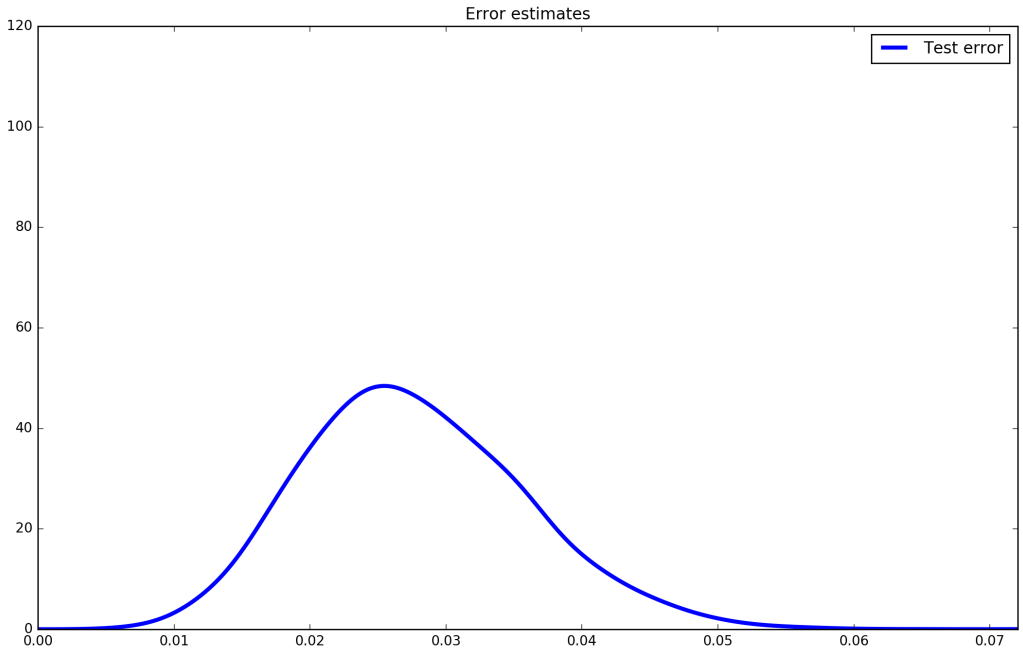

Figure 3.3: Probability distribution of the test error for different samples of the test set in the fifth degree polynomial example.

3.2 Underfitting and Overfitting

If the model is unable to fit the training set, both the training and test errors tend to be high because no hypothesis can be instantiated that accurately reflect the relation between the attributes and the value to predict. This is called *underfitting*. In the case of *underfitting*, replacing the model with one more capable of fitting the data will reduce both training and test errors. This is illustrated in Figure 3.4, as we move from degree 1 to degree 5. However, improving the fit between the model and the training set will eventually begin increasing the *test error*, even though the *training error* decreases. This is called *overfitting*, which is due to the model adapting to details of the training set that do not generalize to the universe from which the data was sampled. Higher degree polynomials have a lower *training error* but a larger *test error*

We can plot the two errors as a function of the degree of the polynomial to see this effect more clearly. Figure 3.5 illustrates this. The *training error*, in blue, decreases steadily as we increase the degree of the polynomial, increasing its ability to fit the training set. However, the *test error*, in red, only decreases until degrees 5 or 6. Afterwards, the models start *overfitting*, increasing the *test error* and the *generalization error*, which is the difference between the *test error* and the *training error*.

3.3 Model Selection and Validation

As Figure 3.5 shows, not all models are equally adequate for finding the hypothesis that best allows us to predict values in our universe $\mathcal{U}$. But, using the estimate of the true error in each case, we can find the model that performs best at this task. This procedure is called *model selection*: we use one set of data, the *training set*, to fit each model. Then we use another set of data to estimate the true error of each hypothesis resulting from fitting each model to the training set. We then select the hypothesis for which this estimate for the true error is lowest. In this case, this would be the polynomial of degree 5 shown in the second panel of Figure 3.4.

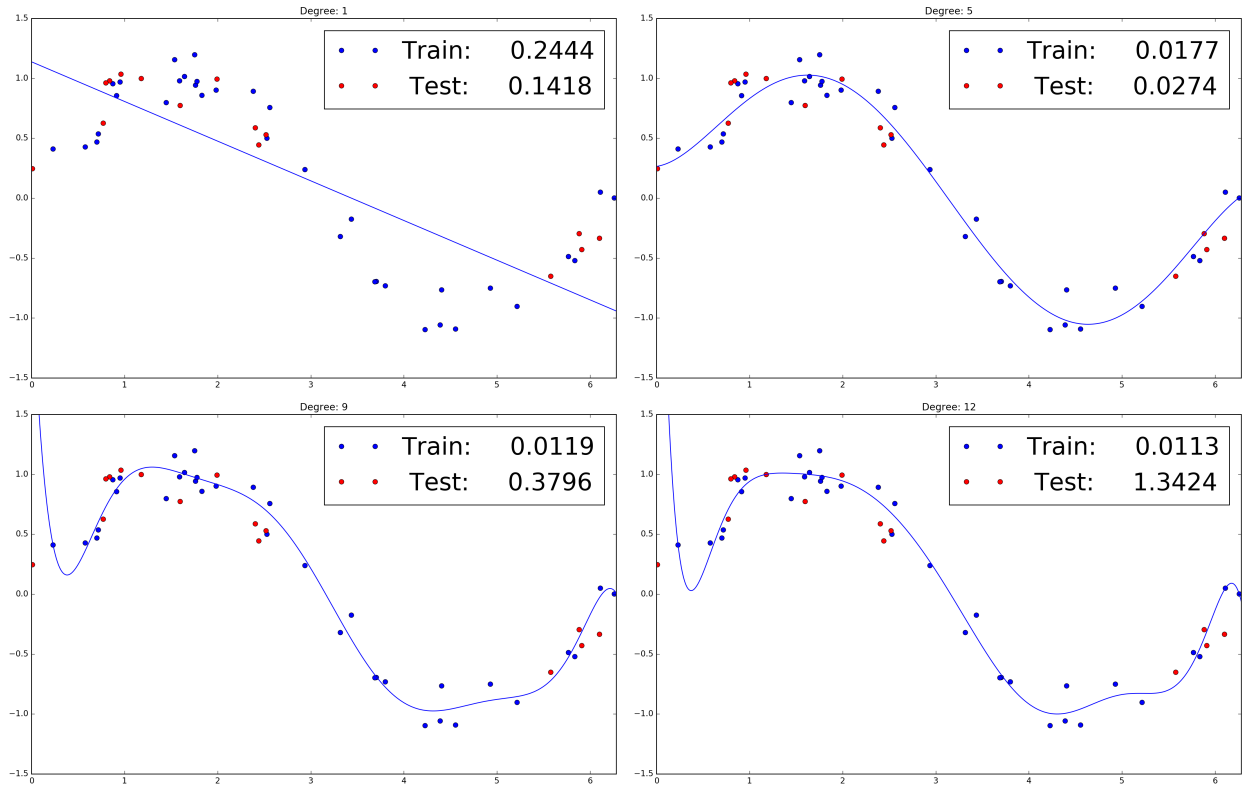

Figure 3.4: Different models fit to the same training set, evaluated with the same test set.

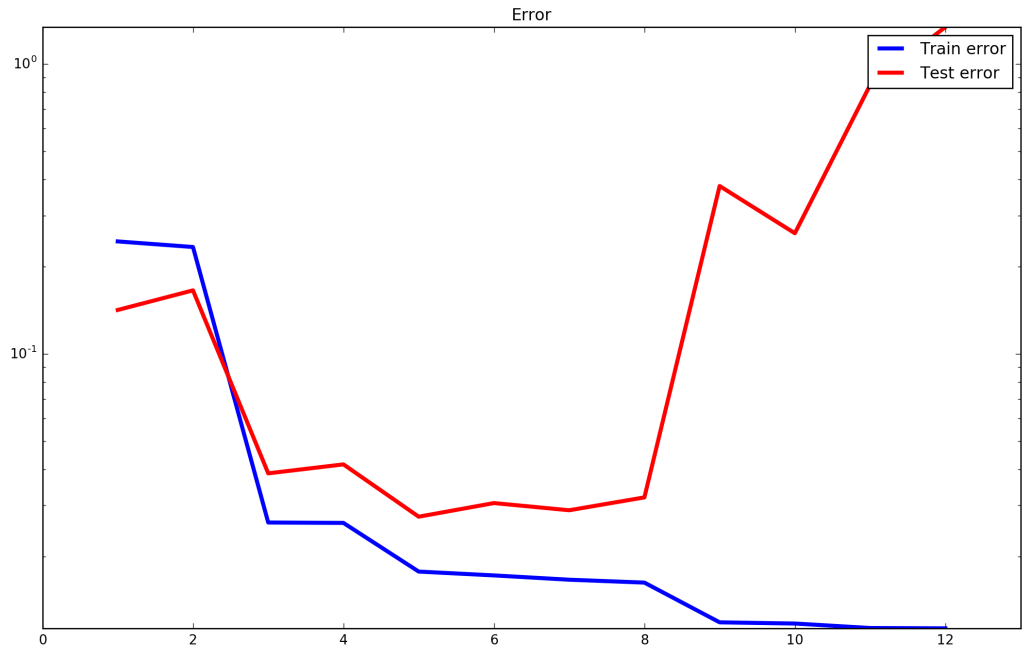

Figure 3.5: Plot of the training error (blue) and the test error (red) as a function of the degree of the polynomial.

However, if we select the hypothesis with the lowest error from a set of error estimates, then this error estimate is no longer an unbiased estimate of the true error. This is because we are selecting the smallest value out of that set of measured errors (one for each model). We can understand this with an analogy. If we choose people at random, some will be taller than average, others will be shorter than average but the average of their height will tend towards the average height of the population.

So, even though the height of people at random is not exactly the average height, it is an unbiased estimator of the average height. This is what happens when we use the *test error* of one hypothesis to estimate its *true error*. However, if we choose people at random in groups and then, from each group, we always pick the shortest person, the average height of those shortest people from each group will no tend towards the average height of the population.

Figure 3.6 compares the unbiased test error distribution, in blue, and the distribution of the smallest error measured in groups of 10 (in red). So, if we use the error estimate to select the best model and hypothesis, then we can no longer use that value as an unbiased estimate of the true error. It will tend to underestimate the true error. This is why, for *model selection* using the error estimates, we need to split our data set in three subsets. The *training set*, to fit each model, the *validation set* to obtain the error estimates to select the best hypothesis, and then a *test set* to obtain a final, unbiased, estimate of the true error. This *test error* can only be used for the final estimate.

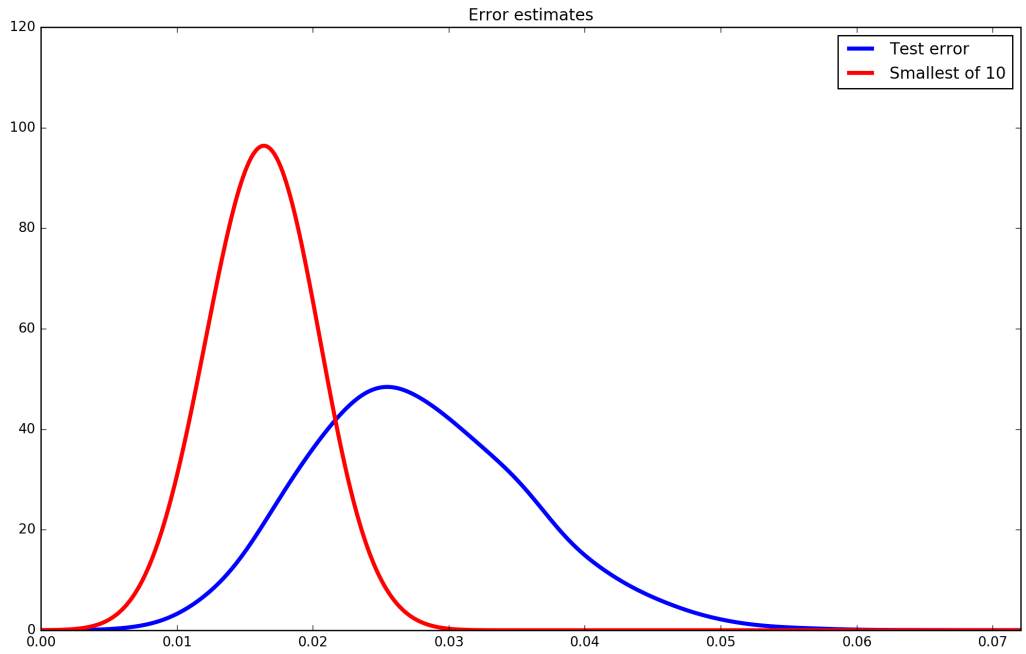

Figure 3.6: The probability distribution of the test error (in blue) and the smallest of groups of 10 test errors.

There are other methods of *model selection*, which we will see later on. In this lecture, the main point is to note this difference between training, validation and testing. Training is the process of fitting the model; validation allows us to choose an hypothesis and testing gives us an unbiased estimate of the true error. To ensure that this final estimate is unbiased, the test set cannot be used at any stage to train hypotheses or select models.

3.4 Regularization

Another approach to solve the *overfitting* problem is to change the learning algorithm to try to prevent the model from adjusting too much to details that do not generalize. One way to do this with our polynomial models is to use a high degree polynomial but add to the error function a penalty as a function of the coefficient values:

$$J(\theta) = \sum_{t=1}^{n} \left[y^t - g(x^t|\theta)\right]^2 + \lambda \sum_{j=1}^{m} \theta_j^2$$

This example, a quadratic penalty function, is called *ridge regression* [12]. Figure 3.7 shows the result of fitting a degree 15 polynomial with different values of the regularization weight λ.

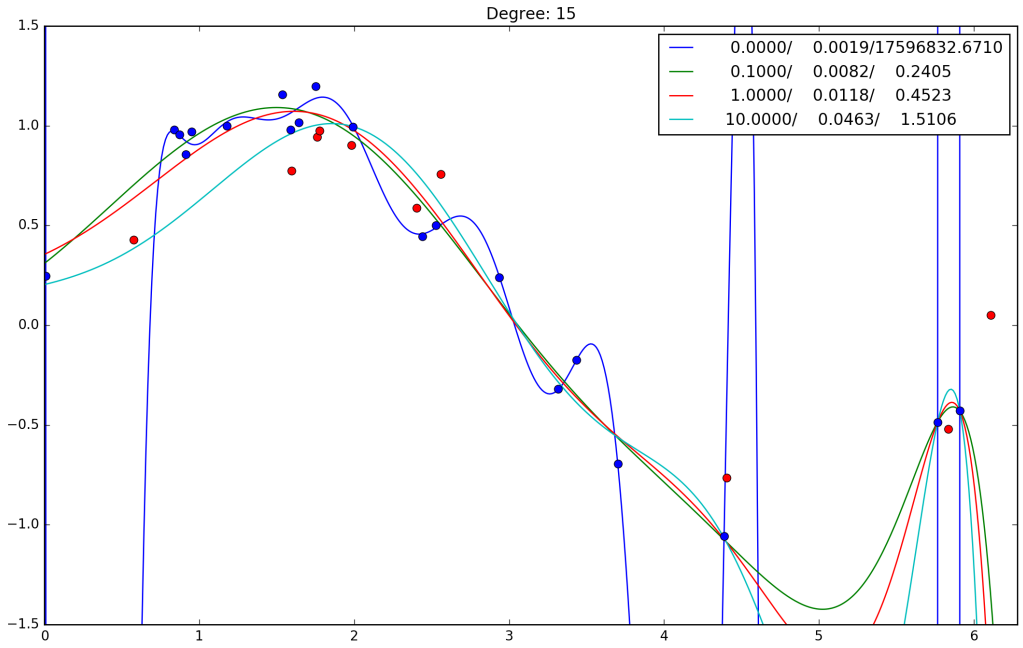

Figure 3.7: Fitting a degree 15 polynomial with different values of λ for regularization. The legend shows the λ, training error and test error.

3.5 Application Example

Figure 3.8 shows the plot of life expectancy versus *per capita* GDP for 180 countries in 2003[1].

In order to find the best model for this data, we will split it randomly into three sets. The *training set*, consisting of half of the points (90 points), the *validation set*, with 45 points, and the *test set*, also with 45 points. This is the code for loading and splitting the data:

```
def random_split(data,test_points):
    """return two matrices splitting the data at random
    """
    ranks = np.arange(data.shape[0])
    np.random.shuffle(ranks)
    train = data[ranks>=test_points,:]
    test = data[ranks<test_points,:]
    return train,test

data = np.loadtxt('life_exp.csv',delimiter='\t')
scale=np.max(data,axis=0)
data=data/scale
train, temp = random_split(data, 90)
valid, test = random_split(temp, 45)
```

[1] http://www.indexmundi.com/g/correlation.aspx?v1=30&v2=67&y=2003&l=en

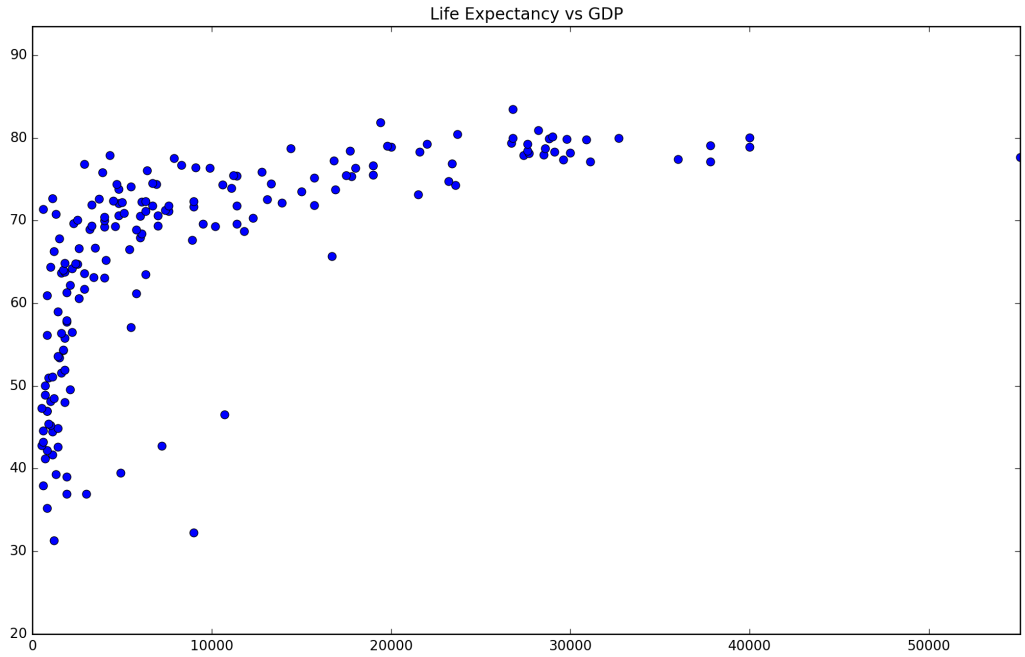

Figure 3.8: Life expectancy versus per capita GDP.

Note that we rescale the data values by dividing them all by the maximum for each column. These maximum values are obtained with the `np.max()` function, specifying the argument `axis=0` to indicate that we want the maximum values computed in the first dimension (the rows). The division of a matrix by a vector is broadcast on the last dimensions (which must match) and, in this case, will divide each row of `data` by the values in `scale`. This rescaling is advisable because large magnitude differences in values can cause instabilities in the polynomial regressions, especially at higher degrees. Now we test different degree polynomials and keep the one with the lowest validation error.

```
def mean_square_error(data,coefs):
    """Return mean squared error
       X on first column, Y on second column
    """
    pred = np.polyval(coefs,data[:,0])
    error = np.mean((data[:,1]-pred)**2)
    return error

best_err = 10000000 # very large number
for degree in range(1,9):
    coefs = np.polyfit(train[:,0],train[:,1],degree)
    valid_error = mean_square_error(valid,coefs)
    if valid_error < best_err:
        best_err = valid_error
        best_coef = coefs
        best_degree = degree

test_error = mean_square_error(test,best_coef)
print best_degree,test_error
```

The result is shown in Figure 3.9, representing the different polynomials.

Once we select the best hypothesis (in this case, the best polynomial with degree 3, with a validation error of 0.0150), we can estimate the true error for this hypothesis using the *test set*. Unlike the

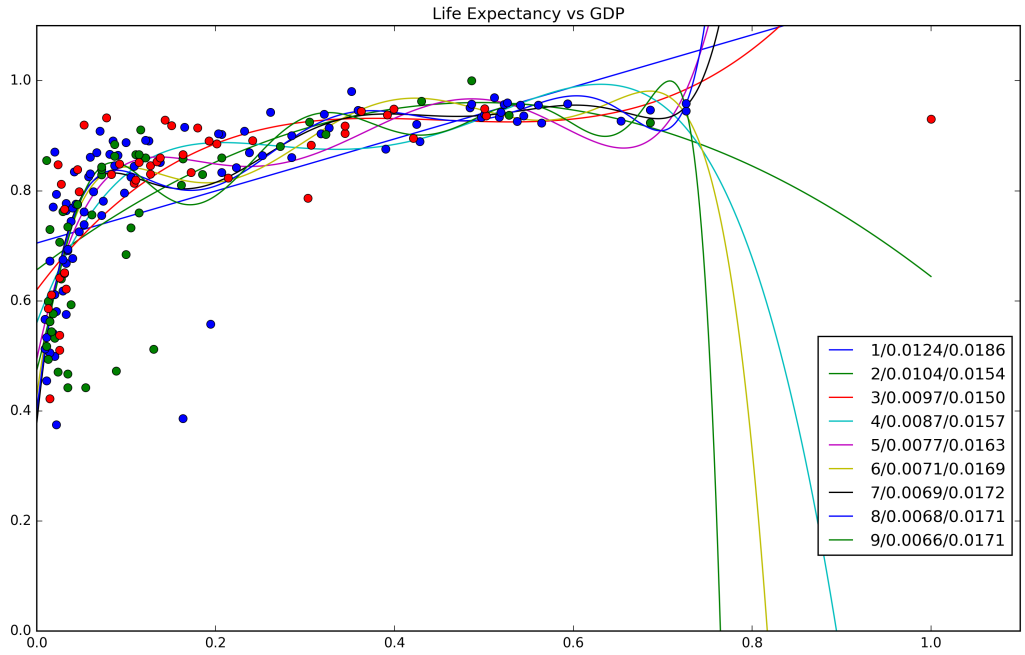

Figure 3.9: Evaluating different models. The training set is represented in blue, the validation set in green and the test set in red. The legend shows the degree of each polynomial and the training and validation errors.

validation error, the test error is an unbiased estimate of the true error because we are not using this value to select any parameter or model. Note, however, that these estimates depend on the random assignment of examples to training, validation and test, and the test error is an estimate of the true error.

Alternatively, we can use regularization with *ridge regression*. The `sklearn` library provides a class `Ridge` (in the `linear_model` module) for *ridge regression*. This is a linear regression solver, but since it is a multivariate regression solver we can expand our data set in order to obtain the same result as a polynomial fit.

First we import the libraries and define the `expand` function to expand the data matrix to a polynomial representation with the specified degree. Then we load the data, rescale the GDP values so they fall into the range [0..1], split into the training, validation and test sets and expand to a polynomial representation of degree 10. In this way, each of our points will have 10 features instead of one.

```
1  import numpy as np
2  from sklearn.linear_model import Ridge
3
4  def expand(data,degree):
5      """expands the data to a polynomial of specified degree"""
6      expanded = np.zeros((data.shape[0],degree+1))
7      expanded[:,0]=data[:,0]
8      expanded[:,-1]=data[:,-1]
9      for power in range(2,degree+1):
10         expanded[:,power-1]=data[:,0]**power
11     return expanded
12
13 orig_data = np.loadtxt('life_exp.csv',delimiter='\t')
14 scale=np.max(orig_data, axis=0)
15 orig_data=orig_data/scale
16 data = expand(orig_data,10)
17 train, temp = random_split(data, 90)
```

```
18 valid, test = random_split(temp, 45)
```

The reason for rescaling is to avoid having numbers of very different orders of magnitude, because we are going to go up to the original values raised to a power of 10. This would cause instabilities in the numeric solver.

Now we try different values of the λ constant (which in the *ridge regression* algorithm is actually designated as α and called `alpha` in the `Ridge` class parameters). We use the `np.linspace()` function to give us a set of evenly spaced values between the minimum and maximum values given. By default, this function returns an array of 50 values, so that is the number of λ values we will try.

```
1  lambs = np.linspace(0.01,0.2)
2
3  best_err = 100000
4  for lamb in lambs:
5      solver = Ridge(alpha = lamb, solver='cholesky',tol=0.00001)
6      solver.fit(train[:,:-1],train[:,-1])
7      ys = solver.predict(valid[:,:-1])
8      valid_err = np.mean((ys-valid[:,-1])**2)
9      if valid_err < best_err:
10         # keep the best
```

If we plot the validation error as a function of the λ constant, we get something like what is shown in Figure 3.10. For more information on the `Ridge` class, consult the Scikit-learn documentation[2].

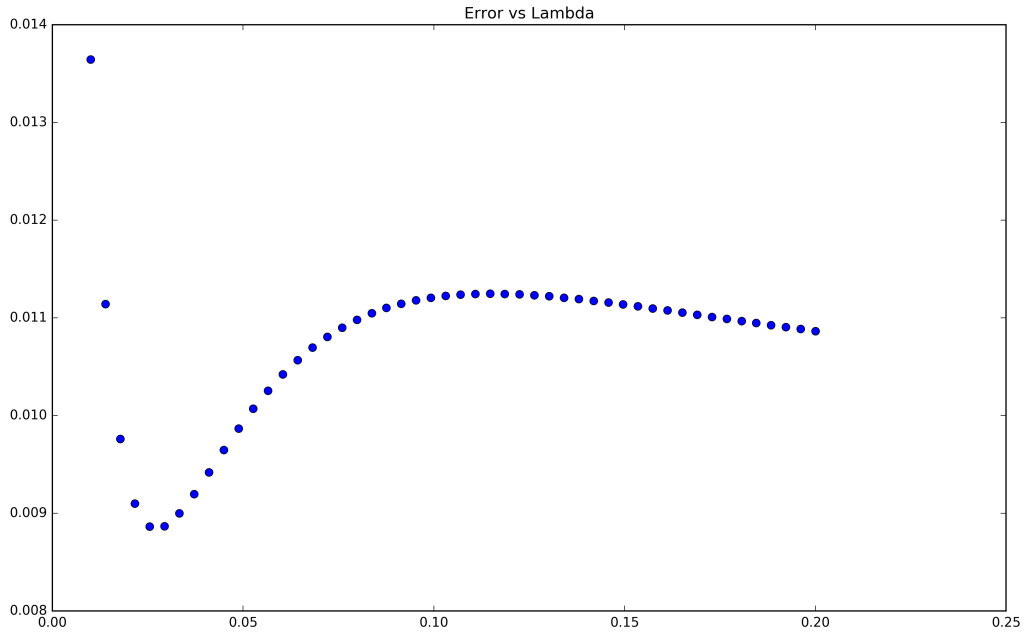

Figure 3.10: Plot of the validation error as a function of the λ constant.

3.6 Summary

The hypothesis that best predicts the target value given the feature vectors of examples from some universe of data is not necessarily the hypothesis that best fits the *training set*. As we improve the fit,

[2] http://scikit-learn.org/stable/modules/linear_model.html

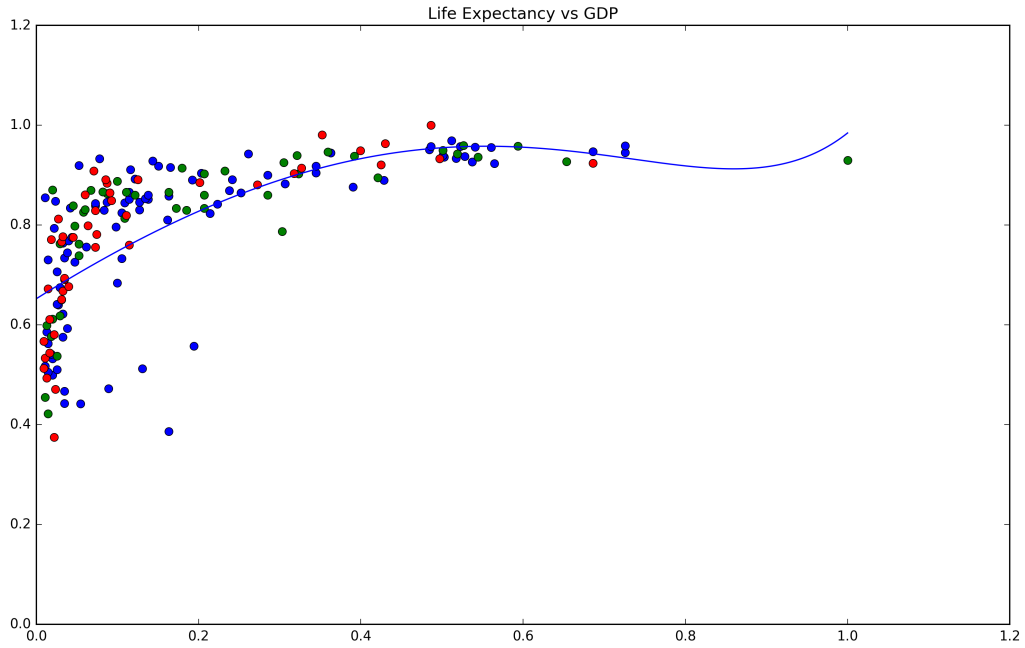

Figure 3.11: Plot of the best line found with the ridge regression. The training set is represented in blue, the validation set in green and the test set in red.

we run into the *overfitting* problem. To solve this, we saw that we can fit our models with a *training set* and use a *validation set* to select the hypothesis that minimizes the error estimated with the *validation set*. However, when we select an hypothesis based on the estimated error, this error estimate will no longer be unbiased, since it influenced our choice. To obtain a better estimate of the true error, we retain a *test set* that we only use for this final estimate.

Later in this course we will revisit *model selection* and explore more sophisticated and reliable ways of selecting the best model. The examples given in this chapter are meant only to illustrate the fundamental aspects of the *overfitting* problem and how to solve it.

3.7 Further Reading

1. Bishop [4], Section 3.1
2. Alpaydin [2], Section 2.6 through 2.8

Chapter 4

Logistic Regression

Classification and linear separability. Normalization and standardization. The logistic regression classifier. Linear separability and dimensionality

4.1 Linear separability

In two dimensions, a pair of sets of points is *linearly separable* if there exists a line that can separate the two sets. Figure 4.3 shows two pairs of sets of points. These are the plots of the activity of gene pairs in tumour (red) and normal tissue (blue). The data is from [1]. In the first panel, the sets are *linearly separable*, as shown by the line dividing them (the *decision boundary*). In the second panel they are not *linearly separable*, as there is no straight line that can divide the two sets.

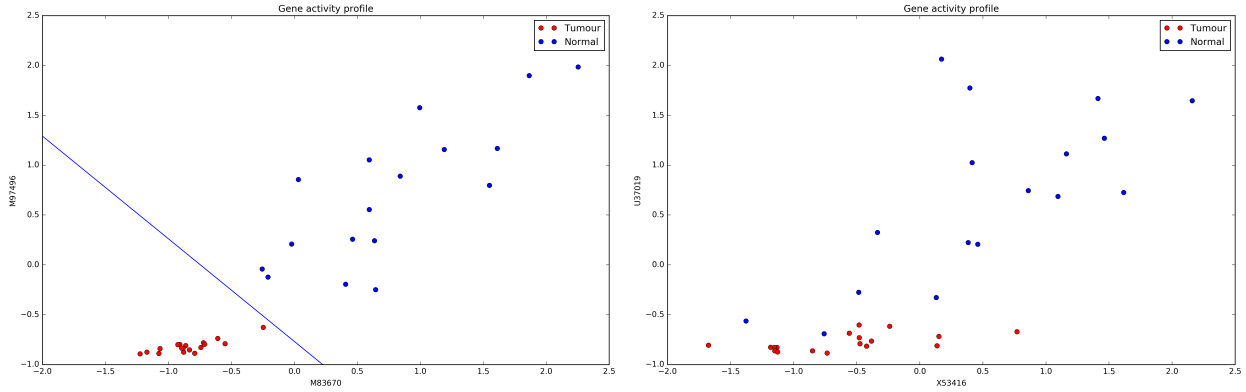

Figure 4.1: The left panel shows a linearly separable pair of point sets. The pair of sets shown on the right is not linearly separable.

In the previous chapters we saw that we can define a straight line with two parameters, in two dimensions. However, for classification we also need to distinguish between the two sides of the line, since we want to separate sets of points with the line as a frontier. We can do this by defining the line with a vector. In this way, the line will consist of all vectors perpendicular to our chosen vector and the vector will indicate a "positive" side of the line. Generalizing for N dimensions, given a vector $\vec{w}$ perpendicular to the desired plane and a constant w_0, the points belonging to the plane can be found by this equation:

$$\vec{w}^T \vec{x} + w_0 = 0$$

Figure 4.2 shows a plane separating two sets of points and the corresponding $\vec{w}$, plotted on the plane. Note that the different axis scales make it appear that the vector is not perpendicular to the plane (but it is).

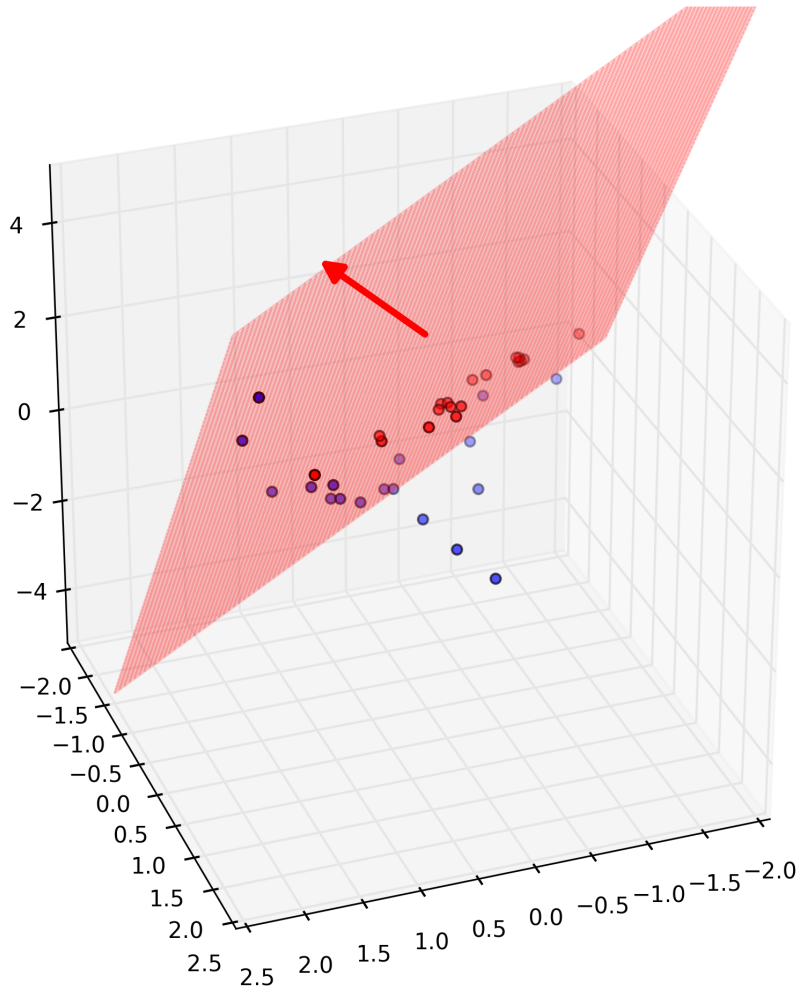

Figure 4.2: A plane separating the two sets of points. The arrow shows the vector normal to the plane (note the distortion due to the different axis scales).

The function

$$y(\vec{x}) = \vec{w}^T \vec{x} + w_0$$

has a positive value on one side of the dividing hyperplane and a negative value on the opposite side. This gives us a way to model a *linear discriminant* separating the two classes. Now we need to find a way to determine the coordinates for $\vec{w}$ and the constant w_0.

4.2 The wrong way: least mean squares

Since the function $y(\vec{x}) = \vec{w}^T \vec{x} + w_0$ is positive on one side of the hyperplane and negative on the other, if we assign a value of 1 to one class and -1 to the other, we can try to find the place of the best dividing hyperplane by minimizing the squared error:

$$E = \sum_{j=1}^{N} (y(\vec{x_j}) - t_j)^2$$

where $\vec{x_j}$ are the points in our training set and t_j the respective class of each point (1 or -1). Note that **this is the wrong way to solve this problem**. However, the result teaches an important lesson about a difference between regression and classification.

To simplify the computation of $y(\vec{x}) = \vec{w}^T\vec{x} + w_0$, we can include w_0 in $\vec{w}$ and simply add a 1 to each $\vec{x}$:

$$\widetilde{w} = (w_0, \vec{w}), \widetilde{x} = (1, \vec{x}), y(\vec{x}) = \widetilde{w}^T\widetilde{x}$$

We will use the data set shown in Figure 4.3, which shows a plot of the activity of the guanylate cyclase activator 2A gene (M97496) versus the activity of the carbonic anhydrase IV gene (M83670) in normal cells (blue) and tumour cells (red) [1]. Our goal is to find the best frontier between these two *linearly separable* sets.

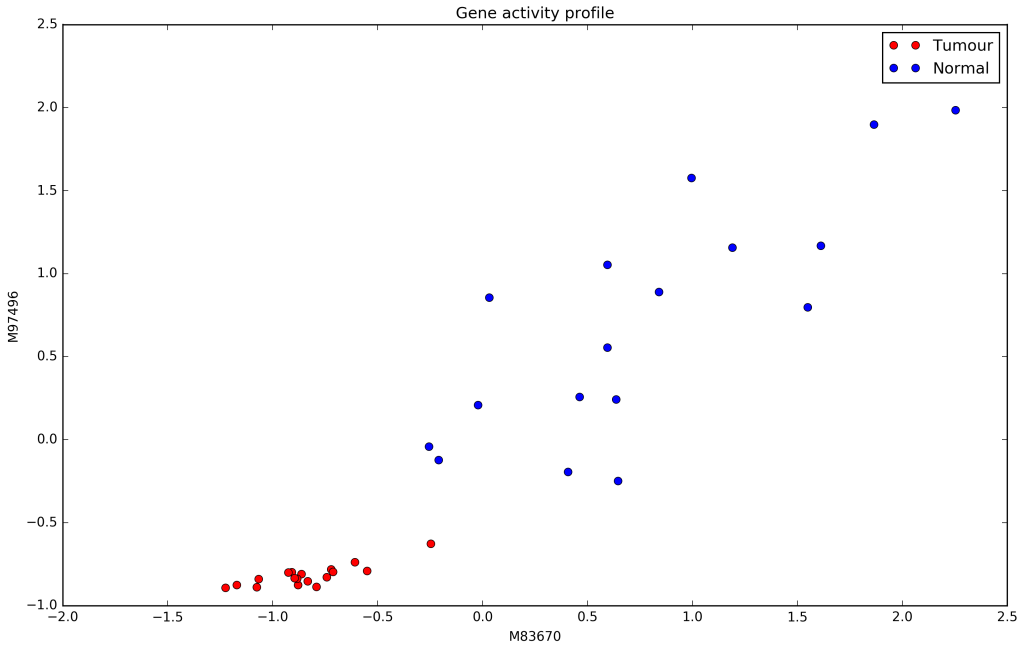

Figure 4.3: Data set for the two classes, tumour and normal, plotting the activity of genes M83670 and M97496 (standardized, see text).

These data is given in a text file, with the activity of each gene and the class, with 0 for normal and 1 for tumour cells:

```
-81     10      1
-30     60      1
...
320   1231      0
172    700      0
...
```

First we read the data and adjust the range of the input values. This is important to bring all values into similar scales in order to prevent the solver from having to deal with different scales. There are two main methods of preprocessing data. *Normalization* is a linear transformation that brings the values of features into the range [0,1]:

$$x_{new} = \frac{x - min(X)}{max(X) - min(X)}$$

where X is the set of all values.

Standardization is a linear transformation that sets the average of the values of features to zero and the standard deviation of these values to 1:

$$x_{new} = \frac{x - \mu(X)}{\sigma(X)}$$

When using real data, we should always do some preprocessing. In this case, we will *standardize* the data. Note that in real applications we must retain these values because we need to process any future examples in exactly the same way as we process the training set, using the same scaling factors.

```
import numpy as np

mat = np.loadtxt(input_data,delimiter='\t')
Ys = mat[:,[-1]]
Xs = mat[:,:-1]
means = np.mean(Xs,0)
stdevs = np.std(Xs,0)
Xs = (Xs-means)/stdevs
```

Now we create that function that adds the column of ones to the input vector, which originally contains two columns for the gene activity. We add this column of ones at the end instead of at the beginning, but the exact placement is indifferent as long as we remember it.

```
import numpy as np

def expand_features(X):
    """append a columns of 1
    """
    X_exp = np.ones((X.shape[0],X.shape[1]+1))
    X_exp[:,:-1] = X
    return X_exp
```

Then the function that computes the quadratic error comparing the signed distance to a frontier line and the classes of the examples. Since we want class values of -1 and 1 but the data originally has classes of 0 for normal and 1 for tumour cells, we need to do some simple algebra to convert this in the last line. Otherwise, we simply compute the inner (dot) product of the two vectors and compute the mean squared error from the result.

```
def quad_cost(theta,X,y):
    """return  error value comparing signed distance with y
       theta is a column of coefficients
       X is a matrix with one example per row, and a 1 in the last column
       y is a vector of classes 0 or 1
    """
    coefs = np.zeros((len(theta),1))
    coefs[:,0] = theta
    vals = np.dot(X,coefs)
    return np.mean((vals-(2*y-1))**2)
```

Now we just add the column of ones and minimize the quadratic error:

```
from scipy.optimize import minimize
```

```
2 import matplotlib.pyplot as plt
3
4 X_exp = expand_features(Xs)
5 coefs = np.ones(X_exp.shape[1])
6 opt = minimize(quad_cost,coefs,(X_exp,Ys),tol=0.00001)
7 coefs = opt.x
8 # plot the chart
```

The result, however, is less than ideal, as shown in Figure 4.4. The reason for this is that point farther from the decision line will have a larger distance value. Since we are minimizing the squared error, the learning algorithm will try to reduce the distance between the decision line and the farthest points. This displaces the decision line away from the actual frontier between the two sets, resulting in some points being misclassified.

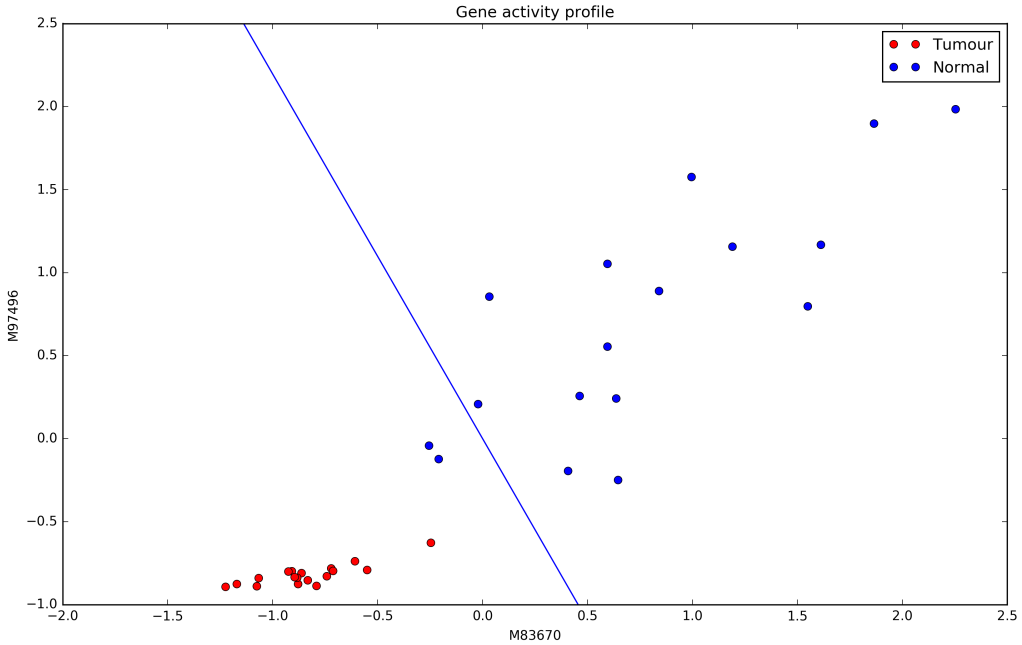

Figure 4.4: Tentative decision line between these two sets computed by minimizing the squared error between the signed distance and the class label. Note how the decision line is incorrectly placed due to the effect of the farthest points.

This is a fundamental difference between regression and classification. In regression problems, it is not desirable to have points distant from the prediction. In classification problems, having points distant from the decision surface (or line, in two dimensions) is not a problem as long as they fall on the correct side of the frontier. So for classification we need a different approach.

4.3 Classification by Logistic Regression

We will be using *logistic regression* as a classifier, with the purpose of obtaining a decision hyperplane that separates different classes. However, *logistic regression* is, at heart, a regression model too, as we will see below.

Let us assume we can find a function of our parameters $\widetilde{w}$[1] and the features $\vec{x}$ that can tell us the

[1]Remember that $\widetilde{w} = (w_0, \vec{w})$.

probability of an example with features $\vec{x}$ belonging to class C_1:

$$g(\vec{x}, \widetilde{w}) = P(C_1|\vec{x})$$

We want to find a decision hyperplane where the probability of finding an example of class C_1 equals the probability of finding an example of class C_0, assuming there are only these two classes.

$$P(C_1|\vec{x}) = P(C_0|\vec{x}) = 1 - P(C_1|\vec{x})$$

which is equivalent to:

$$ln\frac{P(C_1|\vec{x})}{1 - P(C_1|\vec{x})} = 0$$

Thus, we can write

$$ln\frac{P(C_1|\vec{x})}{1 - P(C_1|\vec{x})} = ln\frac{g(\vec{x}, \widetilde{w})}{1 - g(\vec{x}, \widetilde{w})} = \vec{w}^T\vec{x} + w_0$$

Rearranging, we obtain

$$g(\vec{x}, \widetilde{w}) = \frac{1}{1 + e^{-(\vec{w}^T\vec{x} + w_0)}}$$

The function

$$f(x) = \frac{1}{1 + e^{-k(x - x_0)}}$$

is called the *logistic function* and is represented on the right panel of Figure 4.5, for $x_0 = 0$ and $k = 1$.

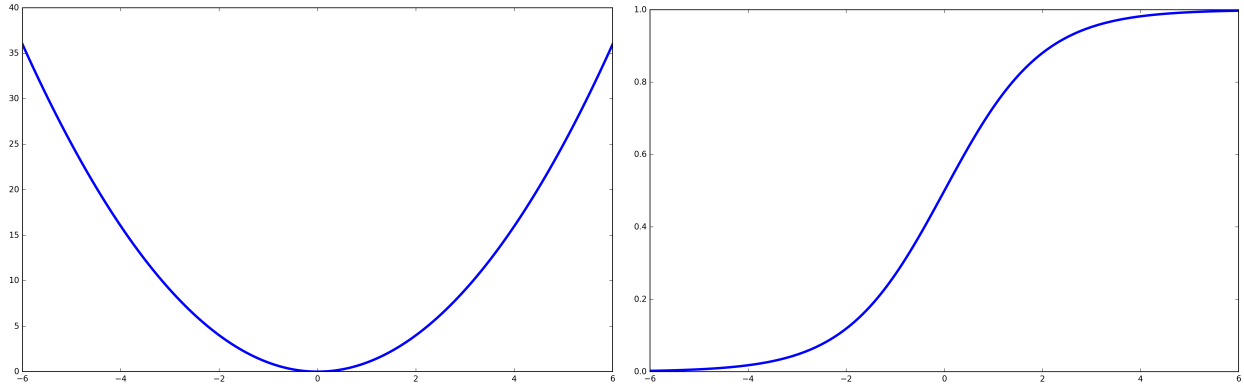

Figure 4.5: Comparing the quadratic and the logistic functions.

This function has the useful feature of varying around a threshold value but being nearly constant away from this threshold. This is what we need to solve the problem we had with the minimization of the squared error, in which the points farther away were having an undesired effect on the decision boundary. Note also that this function attempts to approximate the probability of each point $\vec{x}$ being in class C_1. This is why *logistic regression* is a regression model; when fitting the model we are trying to approximate this continuous probability function. However, because we also choose a cut-off value where we separate the two classes — where $P(C_1|\vec{x}) = P(C_0|\vec{x})$ — we turn this regression model into a classifier. This is a common occurrence in classification, with the classifier model being trained, at bottom, as a regression model. Since the class of hypotheses we are considering in *logistic regression* to separate the classes is the set of all hyperplanes (or planes in 3D and lines in 2D) — *i.e.* linear combinations of the features — this is a linear classifier, when used as a classifier.

Now, given that:

$$g(\vec{x}, \widetilde{w}) = P(C_1|\vec{x})$$

the likelihood of our parameters $\widetilde{w}$, which is the product of the probabilities of the classes of our examples given our hypothesis, will be

$$\mathcal{L}(\widetilde{w}|X) = \prod_{n=1}^{N} \left[g_n^{t_n}(1-g_n)^{1-t_n}\right]$$

and the logarithm of the likelihood is

$$l(\widetilde{w}|X) = \sum_{n=1}^{N} [t_n \ln g_n + (1-t_n)\ln(1-g_n)]$$

where g_n is the result of $g(\vec{x_n}, \widetilde{w})$ and t_n is the true class of point n, which can be 1 or 0.

Thus, the maximum likelihood solution to this problem is the minimum of this cost function:

$$E(\widetilde{w}) = -\sum_{n=1}^{N} [t_n \ln g_n + (1-t_n)\ln(1-g_n)]$$

with

$$g_n = \frac{1}{1+e^{-(\vec{w}^T\vec{x_n}+w_0)}}$$

To implement this, we write the logistic and logistic cost functions:

```
1  def logistic(X):
2      """return logistic function of vector X"""
3      den = 1.0 + np.e ** (-1.0 * X)
4      return 1.0 / den
5
6  def log_cost(theta,X,y):
7      """return  logistic error value
8         X is a matrix with one example per row, and a 1 in the last column
9         y is a vector of classes 0 or 1
10     """
11     coefs = np.zeros((len(theta),1))
12     coefs[:,0] = theta
13     sig_vals = logistic(np.dot(X,coefs))
14     log_1 = np.log(sig_vals)*y
15     log_0 = np.log((1-sig_vals))*(1-y)
16     return -np.mean(log_0+log_1)
```

And then minimize this function instead of the quadratic error. The result is much better, as shown in Figure 4.6.

4.4 Linear separability, revisited

If the classes are not *linearly separable*, it is impossible to find a straight line that can separate the two classes. This is illustrated in Figure 4.7, where the standard logistic regression approach of the last section results in a decision boundary that cannot discriminate between the two classes. In this case,

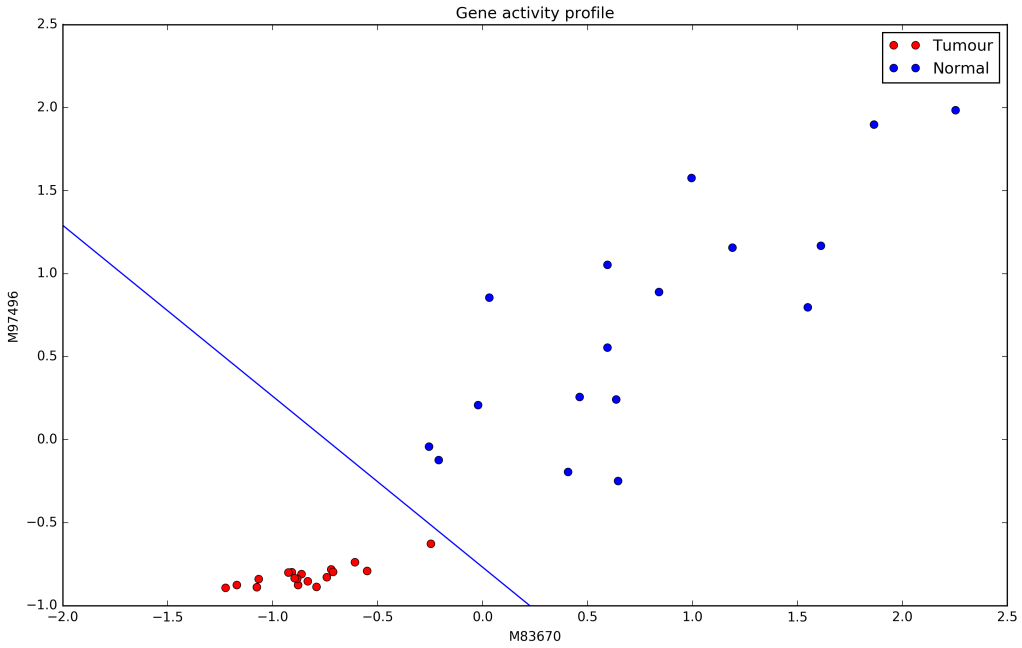

Figure 4.6: Logistic regression on the activity of M83670 and M97496 genes.

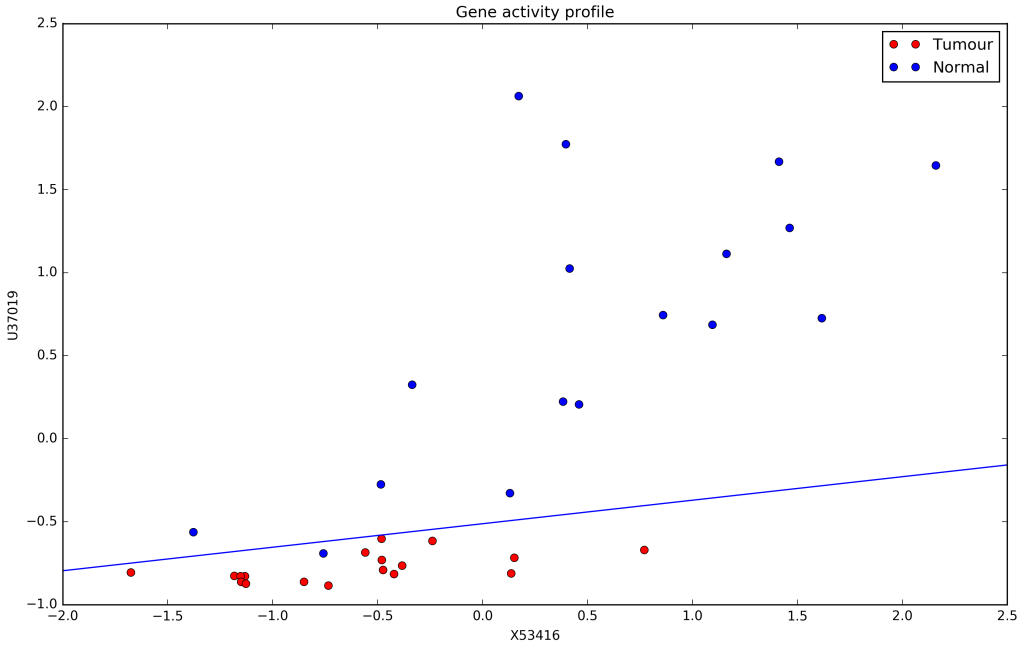

Figure 4.7: Attempting to separate classes that are not linearly separable (activity of X53416 and U37019 genes).

the features are the activity of the actin-binding protein gene (X53416) and the smooth muscle cell calcium binding protein gene (U37019).

However, we can expand our data with an additional feature obtained by (for example) the product of the two original gene activity values. This function is easy to implement.

```
1 def poly_3features(X):
2     """append a column with the product of the two first features
3     """
4     X_exp = np.zeros((X.shape[0],X.shape[1]+1))
5     X_exp[:,:-1] = X
```

```
    X_exp[:,-1] = X[:,0]*X[:,1]
    return X_exp
```

Now we load the data, transform it and use the `LogisticRegression` class from the `sklearn.linear_model` module to find the best plane separating the expanded data.

```
mat = np.loadtxt('gene_data_2.txt',delimiter='\t')
Ys = mat[:,[-1]]
Xs = mat[:,:-1]
means = np.mean(Xs,0)
stdevs = np.std(Xs,0)
Xs = (Xs-means)/stdevs
X_exp = poly_3features(Xs)
reg = LogisticRegression(C=1e12, tol=1e-10)
reg.fit(X_exp,Ys[:,0])
```

Now, our *linear discriminant* will no longer be a line in two dimensions but a plane in three dimensions, corresponding to the expanded data, as shown in Figure 4.8

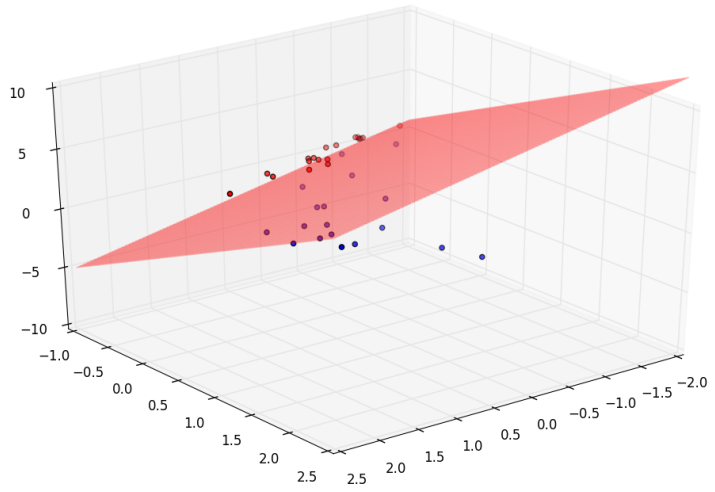

Figure 4.8: Gene activity (X53416 and U37019 genes) data expanded to three dimensions and the decision plane found by logistic regression.

Projecting this decision plane back into the two dimensional space of the original data, we can find the corresponding decision boundary which is no longer a straight line. This is a similar approach to the one we saw with linear regression.

This is still not enough to completely separate the two classes, but we can increase the discrimination power of the classifier by increasing the dimensions used by the logistical regression. For example, using 7 dimensions, as shown in the code below and Figure 4.10.

```
def poly_7features(X):
    """append a five columns with the product,
        square and cube of the first two features
    """
    X_exp = np.zeros((X.shape[0],X.shape[1]+5))
```

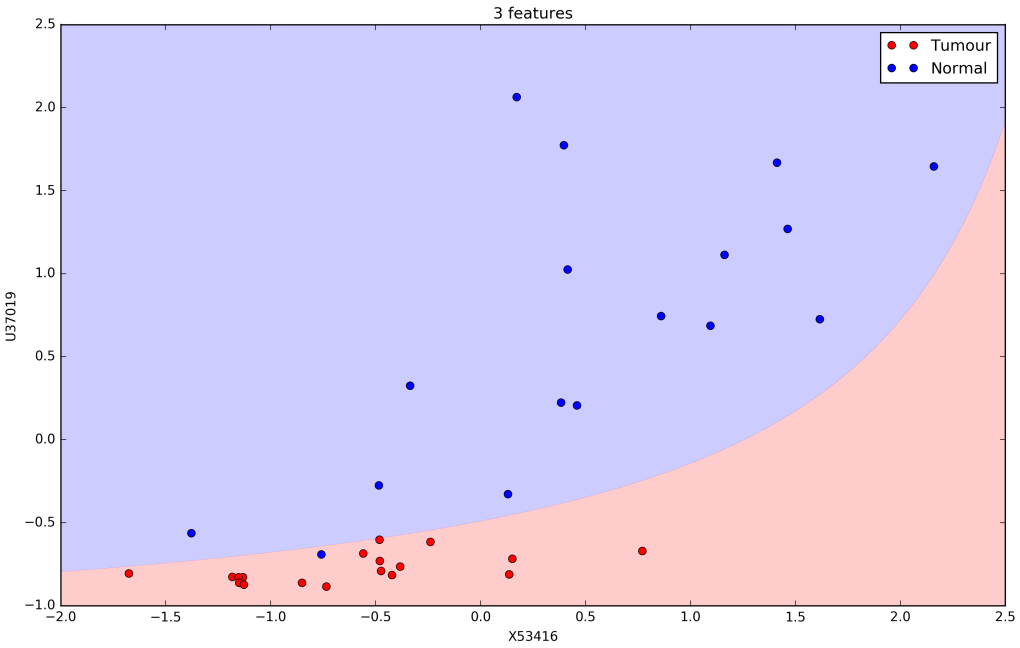

Figure 4.9: Gene activity (X53416 and U37019 genes), decision boundary obtained using a third feature value obtained from the product of the first two.

```
6       X_exp[:,:-5] = X
7       X_exp[:,-3] = X[:,0]*X[:,1]
8       X_exp[:,-2] = X[:,0]**2
9       X_exp[:,-1] = X[:,1]**2
10      X_exp[:,-5] = X[:,0]**3
11      X_exp[:,-4] = X[:,1]**3
12      return X_exp
```

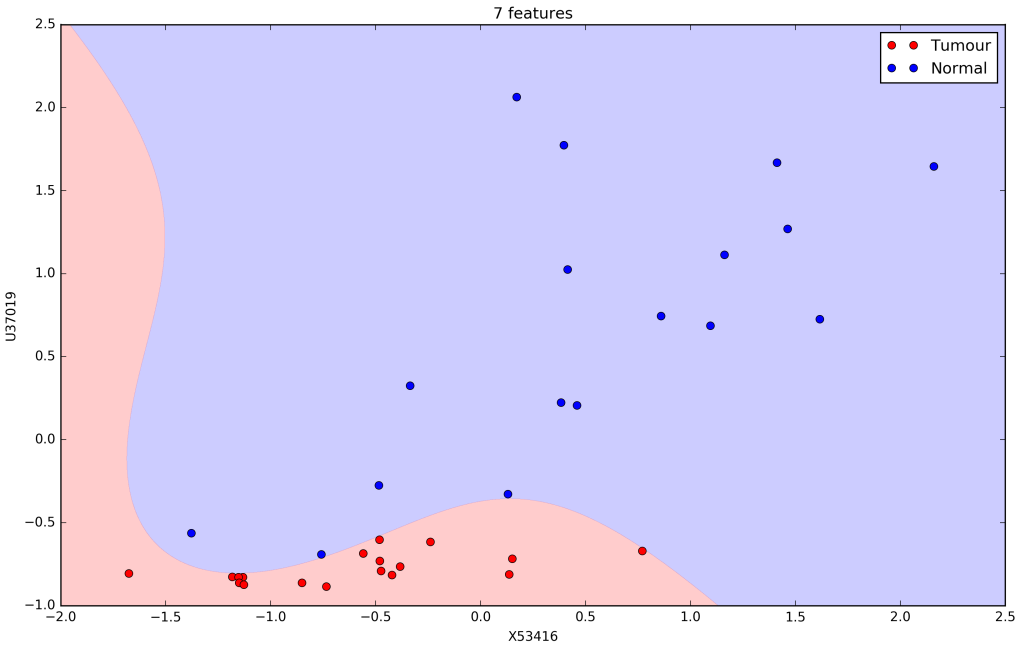

Figure 4.10: Gene activity (X53416 and U37019 genes), decision boundary obtained using a five additional features computed from the original two.

Once again, we need to face the possibility of overfitting our data. This will be covered in the next lecture.

4.5 Summary

In this chapter we covered the notions of *linear separability* and *linear discriminants*. We also saw that finding linear discriminant by minimizing a simple quadratic error did not place the boundary in the right position. However, we saw that using a logistic function to model the probability of obtaining examples of each class as a function of the feature vectors and the discriminant allows us to find the best discriminant by maximum likelihood. We also saw how expanding our features into a feature space with more dimensions can allow us to use linear discriminants in higher dimensions to separate classes that are not linearly separable in the original feature space.

4.6 Further Reading

1. Bishop [4], Sections 4.1.1, 4.1.3 and 4.3.2

Chapter 5

Overfitting Logistic Regression

Classification errors. Cross validation. Model selection with cross validation and Logistic Regression. Regularization in Logistic Regression

5.1 Scoring binary classifiers

In Chapter 3, we saw the difference between the *training error*, measured on the set of points used to fit the model; the *validation error*, measured outside the training set to estimate the error of each of a number of hypotheses in order to select the best one; and the *test error*, measured in an another set of points and remaining an unbiased estimator of the *true error* because it is never used to fit or select an hypothesis. In all these cases, we always measured the quadratic error between the predicted and the target values:

$$E(\theta|\mathcal{X}) = \sum_{t=1}^{n} [y^t - g(x^t|\theta)]^2$$

In regression, we used this function to fit the data, validate and test the regression hypotheses. However, in Chapter 4, we saw that, for classifying data using a linear discriminant defining a hyperplane, the quadratic error measured as the distance to the discriminant was not the best cost function for minimization. In Logistic Regression, we used a logistic function to estimate the probability of each class and then obtained, by maximum likelihood, a cost function to minimize:

$$E(\widetilde{w}) = -\sum_{n=1}^{N} [t_n \ln g_n + (1 - t_n) \ln(1 - g_n)]$$

with

$$g_n = \frac{1}{1 + e^{-(\vec{w}^T \vec{x_n} + w_0)}}$$

Since g_n is our predicted probability of example n belonging to class $t = 1$, this is actually the cross-entropy between the probability distribution of our data and the probability distribution of our predictions. Averaging over all samples, we get the average cross-entropy. This is called the *logistic loss* or *log loss* function:

$$L(\widetilde{w}) = \frac{1}{N} \sum_{n=1}^{N} H(p_n, q_n) = -\frac{1}{N} \sum [t_n \ln g_n + (1 - t_n) \ln(1 - g_n)]$$

The lower the *log loss*function the better our hypothesis is at predicting the training data.

Another possible measure is the quadratic error between the probability prediction given by our hypothesis g_n and the class $t \in 0, 1$. This is called the *Brier score*:

$$E(\widetilde{w}) = \frac{1}{N}\sum_{n=1}^{N}[t_n - g_n]^2$$

Figure 5.1 shows the surface of the predicted probabilities of each point belonging to class 1 and the points used to fit this model. The quadratic error will be the sum of the squared differences between this surface and the class value for each point. Note that, in this case, the error is measured not from the distance to the frontier but from the difference between the class and the estimated probability of the point belonging to class 1.

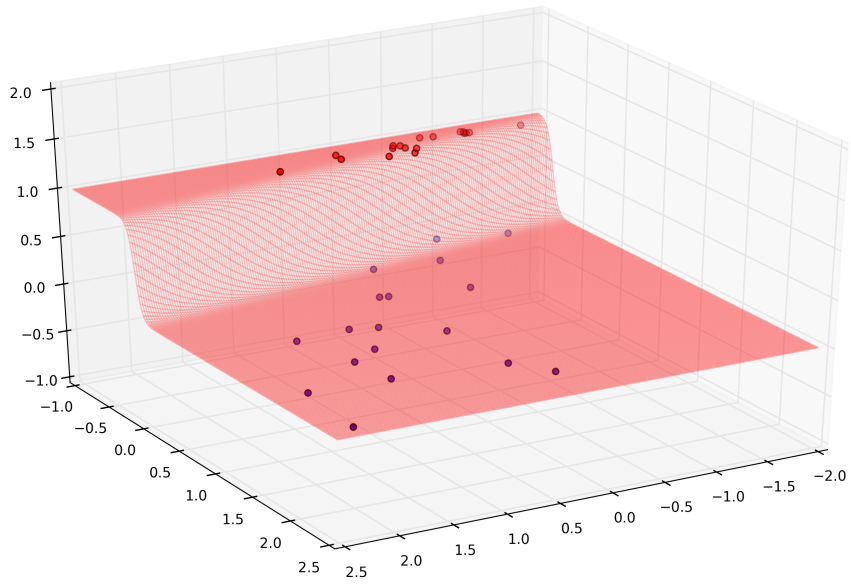

Figure 5.1: Surface representing the predicted probability and the points plotted in their classes with z=0 or z=1.

Another possible error measure is the *accuracy* of the classifier. Let us suppose we consider that any point with $g_n \geq 0.5$ is predicted to be in class $t = 1$ and any point with $g_n < 0.5$ is predicted to be in class $t = 0$. We can consider four different possibilities:

1. *True positive*: the example belongs to class 1 and was predicted to belong to class 1.
2. *False positive*: the example belongs to class 0 and was predicted to belong to class 1.
3. *True negative*: the example belongs to class 0 and was predicted to belong to class 0.
4. *False negative*: the example belongs to class 1 and was predicted to belong to class 0.

Schematically, we can represent these four possibilities with a *confusion table*:

		Examples	
		Class 1	Class 0
Predictions	Class 1	True Positive	False Positive
	Class 0	False Negative	True Negative

The *accuracy* of a binary classifier over a set of N points is thus:

$$accuracy = \frac{true\ positives + true\ negatives}{N}$$

Considering this classification, we can also define the *precision* and *recall* of the classifier:

$$precision = \frac{true\ positives}{true\ positives + false\ positives} \qquad recall = \frac{true\ positives}{true\ positives + false\ negatives}$$

In other words, *precision* is the fraction of correctly classified positive examples in all examples classified as positive (correctly or not), whereas *recall* is the fraction of correctly classified positive examples from the set of all positive examples. This gives us another useful measure of the performance of a classifier, the *F1* score, which is the harmonic mean of *precision* and *recall*:

$$F1 = \frac{2 \times true\ positives}{2 \times true\ positives + false\ positives + false\ negatives}$$

$$F1 = 2\frac{precision \times recall}{precision + recall}$$

Although the usual approach is to consider that $g_n \geq 0.5$ predicts a point in class 1 (positive), we can change the value of this threshold and consider a more general approach of predicting the positive class at $g_n \geq \alpha,\ \alpha \in [0, 1]$. Figure 5.2 shows the effect of drawing the frontier at different values of α and then plotting the number of true and false positives as a function of α. If α is too small, all points will be classified as being in the positive class, so there will be a maximum number of true and false positives. As α increases, the false positives should start decreasing first. When α is too high, then all points are classified as being in the negative class, which means there are no false positives but no true positives either.

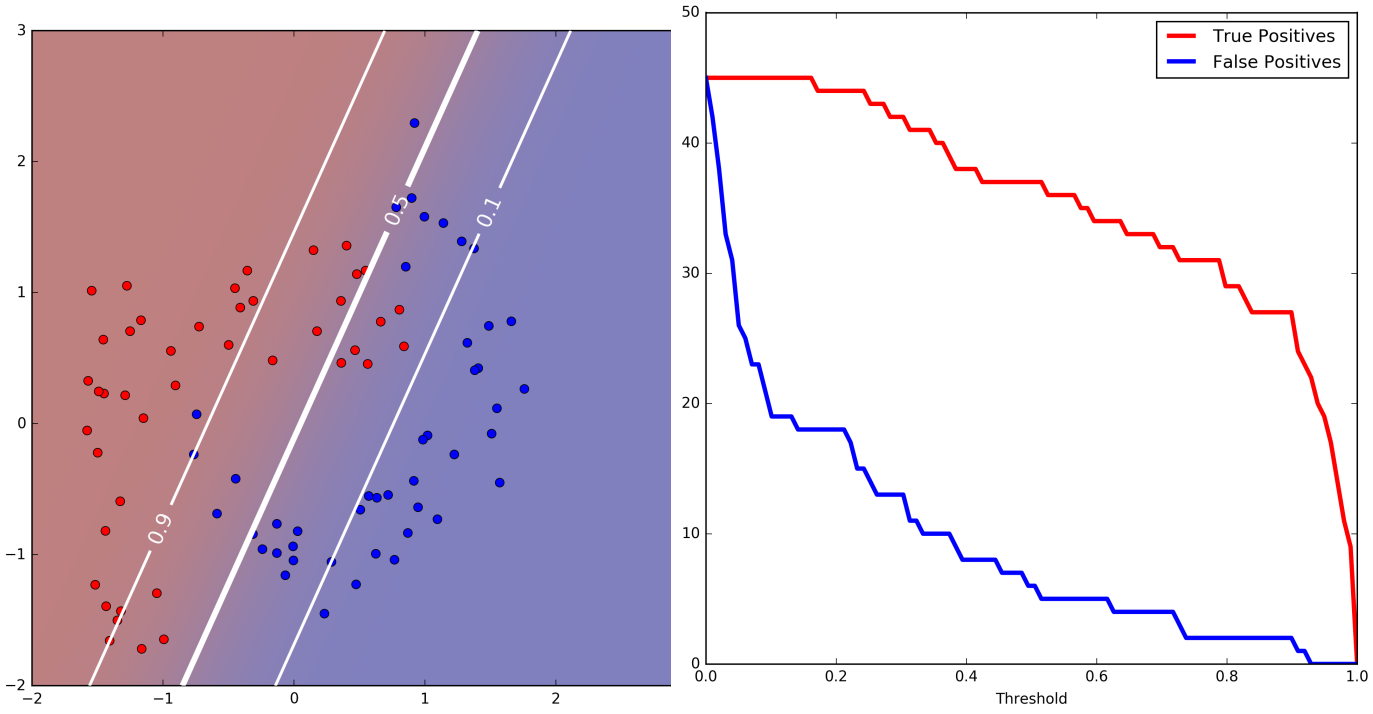

Figure 5.2: The left panel shows different contours of the probability of finding class 1 (in red). The right panel shows the true positives and false positives as a function of the threshold α.

Using this variation in the fraction of true positives and false positives as a function of the threshold, we can also evaluate a binary classifier by plotting a *receiver operating characteristic* curve, or *ROC* curve[1]. The *ROC* curve is plotted by computing the fraction of *true positives* and *false positives* at different score thresholds. A classifier performs all the better the greater the fraction of *true positives* relative to the *false positives* for different threshold levels. In other words, the larger the area below the *ROC* curve the better the classifier's performance. Figure 5.3 shows an example of a *ROC* curve.

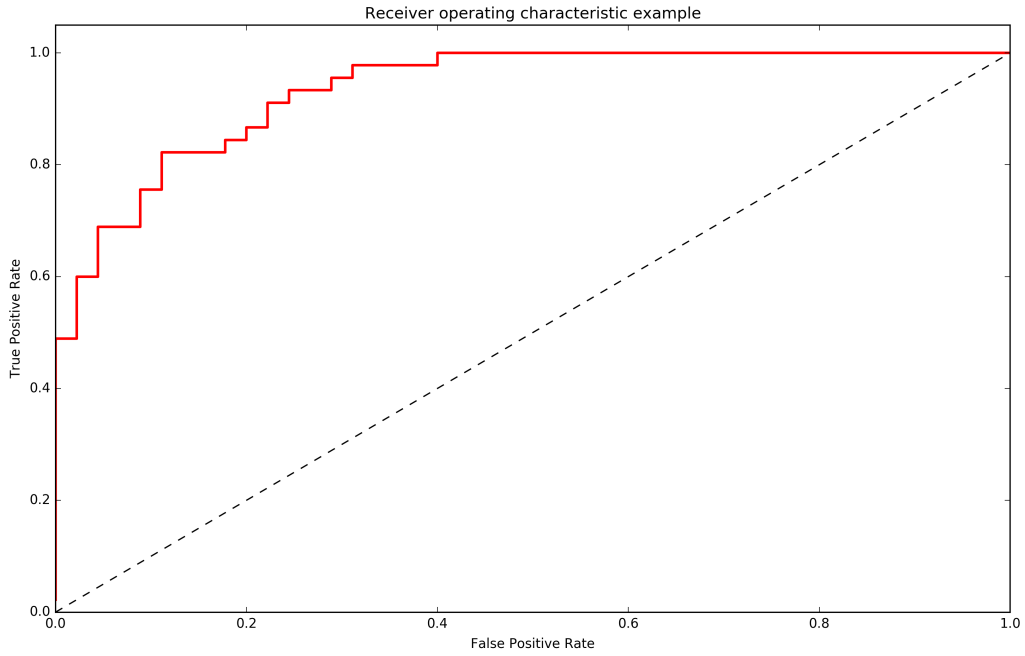

Figure 5.3: A ROC curve.

Classifiers in the Scikit-Learn offer a `score(X,Y)` method that returns the *accuracy* score for the classifier computed on the given data. From this score, we can also compute the error as one minus the *accuracy*.

```
1 from sklearn.linear_model import LogisticRegression
2
3 reg = LogisticRegression()
4 reg.fit(X_r,Y_r)
5 test_error = 1-reg.score(X_t,Y_t)
```

5.2 Cross-Validation and Model Selection

In Chapter 3 we saw a simple way to solve the overfitting problem, which was to select the hypothesis that had a smaller *validation error*. To do this we split our data set into a *training set* and *validation set* (and, if desired, a *test set* to estimate the *true error* of the selected hypothesis). However, all these estimates are random samples from some probability distribution and we can improve them by averaging over several repetitions. Furthermore, doing validation in that way only allowed us to evaluate specific hypotheses and not the models themselves. Cross-validation solves these problems.

[1]The name comes from the original use of this method, which was to optimize the detection rate of aircraft in radar signals during the second world war

To do cross-validation, we partition our data into a number of disjoint *folds*. For example, if we have 50 points and want to use 5-fold cross-validation, we place 10 points into each fold. Then we train our model with all folds but one, validate on the fold that was left out, and repeat for all folds. In the end we average the validation error and this gives us an estimate of the true error that, on average, hypotheses generated from this model will have on this type of data. Figure 5.4 shows an example of 5-fold cross validation using the gene expression data. Each panel shows an hypothesis obtained by fitting the model to four of the folds (indicated by the smaller points) and then validating using the fold left out.

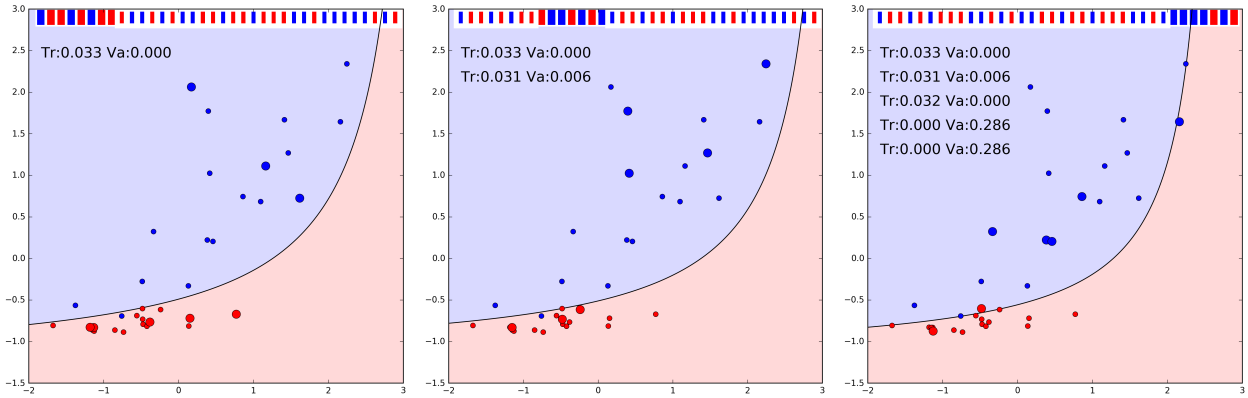

Figure 5.4: Example of 5-fold cross validation, showing the plots for folds 1, 2 and 5. In each panel, one of the folds is left out for validation, the other folds are used for training. The larger points are those used for validation in each fold. The training and validation errors are kept for each fold and then averaged in the end.

In general, *k-fold cross-validation* can be done with any number of folds from two to the number of data points. In this last case, it is called *leave-one-out cross-validation*.

To illustrate this, consider the data set in Figure 5.5. We want to find the best model to separate these data with a logistic regression. First, we load the data, shuffle the order of the points randomly, and then we set aside a third of the data points for the final error evaluation (the *test set*). Ordering the points at random is often necessary to eliminate any correlations in the data set. For example, in this case, all positive class examples are first in the file. We also standardize the features.

```
1  import numpy as np
2
3  mat = np.loadtxt('dataset_90.txt',delimiter=',')
4  ix = list(range(mat.shape[0]))
5  np.random.shuffle(ix)
6  mat_shuf = mat[ix,:]
7  Ys = mat_shuf[:,[0]]
8  Xs = mat_shuf[:,1:]
9  means = np.mean(Xs,0)
10 stdevs = np.std(Xs,0)
11 Xs = (Xs-means)/stdevs
```

We will select the best model using 10-fold cross-validation on the training set. The different models to consider are different expansions of the original data, $x_1, x_2, x_1, x_2, x_1 \times x_2, x_1, x_2, x_1 \times x_2, x_1^2, ...$, each resulting in a model with a different number of features. To do this, we expand the original features polynomially into a matrix of 16 features. Please note that this is not a common use of logistic regression. Explicitly expanding features like this is not very efficient; there are better algorithms for

this that we will see later. However, this exercise is useful to help understand the idea of transforming the examples so that they become linearly separable.

```
def poly_16features(X):
    X_exp = np.zeros((X.shape[0],X.shape[1]+14))
    X_exp[:,:-14] = X
    X_exp[:,-14] = X[:,0]*X[:,1]
    X_exp[:,-13] = X[:,0]**2
    X_exp[:,-12] = X[:,1]**2
    #... rest of the expansion here
    return X_exp
```

As we saw previously, the larger the dimension into which we expand the original data, the easier it is to separate the classes in the training set but the more likely the model is to overfit the data. So, we partition the training set into 10 folds, train each model 10 times, leaving out one fold for validation and average the training and validation error. In this case, we estimate the error using the *Brier score*, which is the average square difference between the class value and the predicted probability of each point being in class 1. This is easy to do with the `sklearn` library. First we create a function that returns the training and test error given a data set and the indexes of the training and test points, using the number of features indicated.

```
from sklearn.linear_model import LogisticRegression

def calc_fold(feats, X,Y, train_ix,test_ix,C=1e12):
    """return classification error for train and test sets"""
    reg = LogisticRegression(penalty='l2',C=C, tol=1e-10)
    reg.fit(X[train_ix,:feats],Y[train_ix,0])
    prob = reg.predict_proba(X[:,:feats])[:,1]
    squares = (prob-Y[:,0])**2
    return (np.mean(squares[train_ix]),
            np.mean(squares[test_ix]))
```

This function fits the logistic regression classifier to the training set, then predicts the probabilities for all the set and returns the mean squared error for the training and test sets. Note that the computation of the Brier score by subtracting the predicted probability and the class assumes that classes are 0 and 1. If the class labels are not 0 and 1 we must convert them to these values.

Now we use the `KFold` class to generate an iterator for the training and validation sets. Here is an example of how a `Kfold` object works:

```
from sklearn.model_selection import KFold

x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
kf = KFold(n_splits=4)
for train, valid in kf.split(x):
    print (train, valid)

[ 3  4  5  6  7  8  9 10 11] [0 1 2]
[ 0  1  2  6  7  8  9 10 11] [3 4 5]
[ 0  1  2  3  4  5  9 10 11] [6 7 8]
[ 0  1  2  3  4  5  6  7  8] [ 9 10 11]
```

We expand the data to the maximum number of features, leave out one third of the data for testing and then loop through the range of features. For each model, we iterate through the 10 different folds to do the cross-validation, printing the average training and validation errors:

```
from sklearn.model_selection import train_test_split, StratifiedKFold

Xs=poly_16features(Xs)
X_r,X_t,Y_r,Y_t = train_test_split(Xs, Ys, test_size=0.33, stratify = Ys)
folds = 10
kf = StratifiedKFold(n_splits=folds)
for feats in range(2,16):
    tr_err = va_err = 0
    for tr_ix,va_ix in kf.split(Y_r,Y_r):
        r,v = calc_fold(feats,X_r,Y_r,tr_ix,va_ix)
        tr_err += r
        va_err += v
    print(feats,':', tr_err/folds,va_err/folds)
```

Figure 5.5 illustrates the ten hypotheses obtained for each of two models, with 2 and 6 features, and the mean training and validation errors.

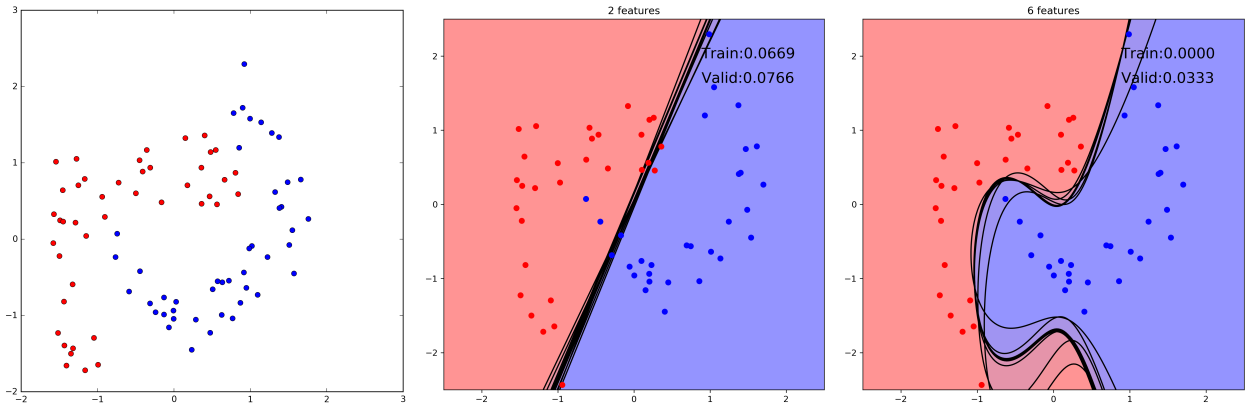

Figure 5.5: Example dataset. The first panel shows the full data set, with positive class (1) in red, negative class (0) in blue. The next two panels show the training set, used in cross-validation. The lines show the 10 different hypotheses obtained during the 10-fold cross-validation. The training and validation errors are the average for the 10 folds.

Figure 5.6 shows the average training and validation error measured for the set of models considered, with expansions of up to 15 features. In this case, we can see that the best model appears to be the one with 9 features. We now train this model with all the points in the training set and estimate the *true error* with the *test set*. Even though the cross-validation error, by itself, is not biased, since we used cross-validation to select the best model we now need to estimate the true error of the best model with examples outside the training set. The hypothesis obtained from the best model trained on the whole training set and the error estimate obtained with the test set is shown on the right panel of Figure 5.6.

5.3 Cross-Validation and Regularization

We can also use cross-validation to find parameters, such as when optimizing regularization. The `LogisticRegression` class can take a regularization parameter, `C`, which, when using a L2 type

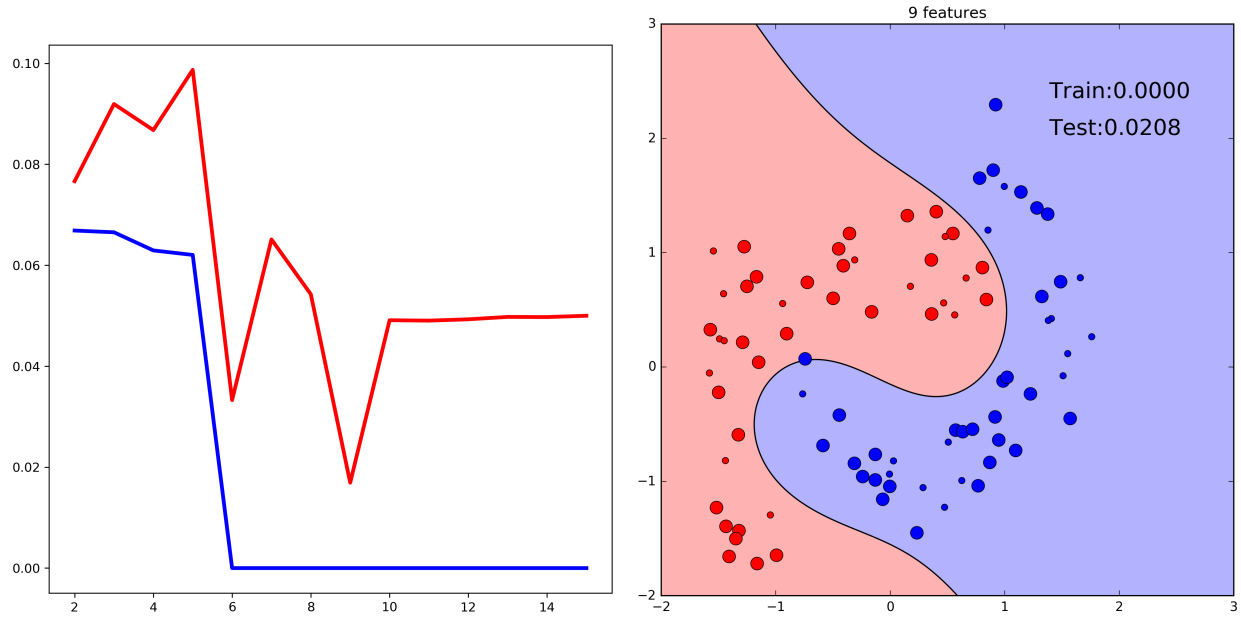

Figure 5.6: The left panel shows the average training and validation error as a function of the number of features in the model. The right panel shows the best model (9 features) trained with the whole training set and tested with the test set (larger points).

regularization, as we saw before with *ridge regression*, corresponds to $\frac{1}{\lambda}$. So our regularization term in this case will be

$$\frac{1}{C}\sum_{j=1}^{m} w_j^2$$

where w_j are the coefficients for the hyperplane separating the classes.

Penalizing the hyperplane will force the coefficients of $\widetilde{w}$ to be smaller. This affects the slope of the logistic function:

$$g(\vec{x}, \widetilde{w}) = \frac{1}{1 + e^{-\widetilde{w}^T \vec{x}}}$$

Without regularization, $\widetilde{w}$ can be as large as necessary and the logistic function can be very sharply sloped, allowing the discriminant to be placed very close to the data points. In this example below we can see that, without regularization, the logistic function is steep enough to separate these classes perfectly:

However, this is probably over-fitting the data since the sole blue point close to the red class is likely to be an outlier and not representative of the data in general. If we regularize the logistic regression classifier, the regularization will force $\widetilde{w}$ to be smaller and thus decrease the slope of the logistic function. This forces the margins around the discriminant to be wider, which in turn forces the discriminant away from the larger number of red class points:

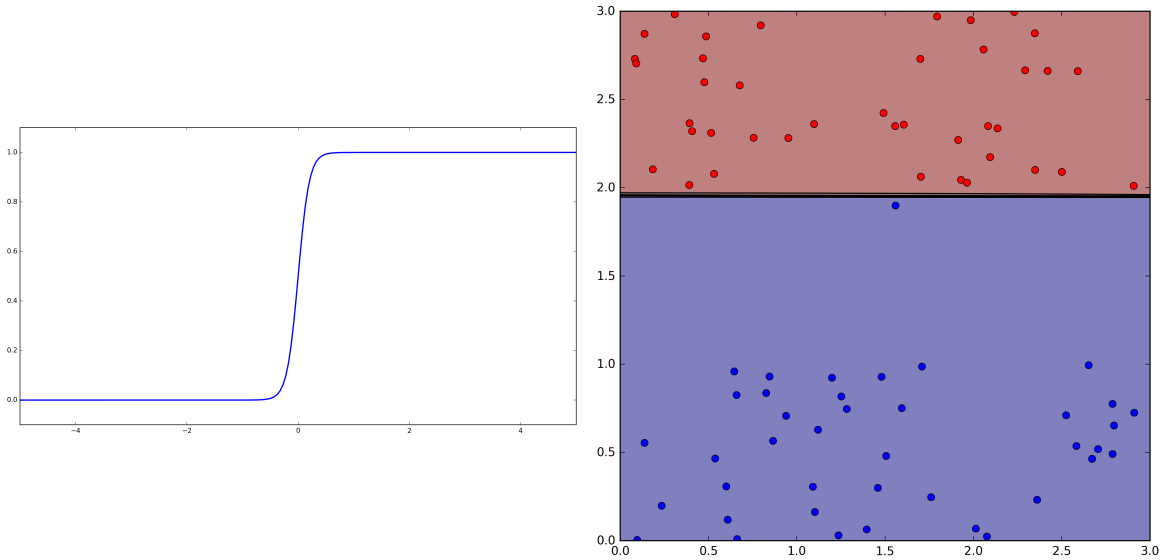

Figure 5.7: Arbitrarily steep logistic function, without regularization.

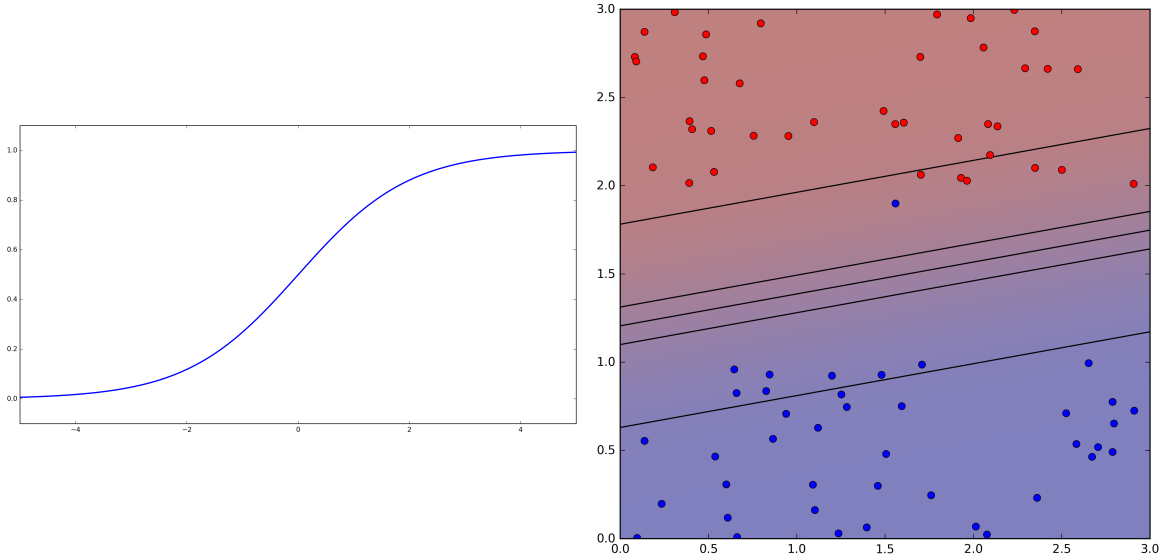

Figure 5.8: Regularization reduces the size of coefficients, making the logistic function less steep.

Figure 5.9 shows the average training and validation error measured for different values of `C`, from 10^{-5} to 10^{15}, always using the model with 15 features, for the original data set shown in Figure 5.5. The right panel shows the result of fitting the model using a `C` value of 10^{5} to all the points in the training set and then testing with the test set we left out in the beginning.

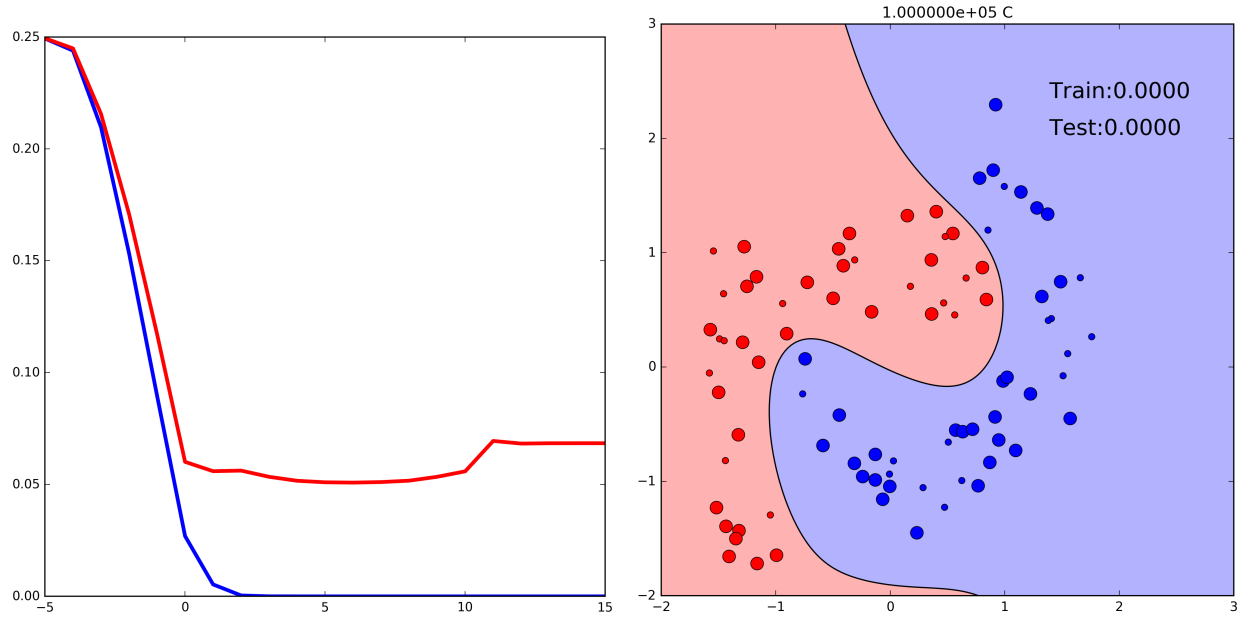

Figure 5.9: The left panel shows the cross-validation error plotted against $log_{10}(C)$ for the 15 features model. The right panel shows the 15 features model regularized with $C = 10^5$ trained on the complete training set.

5.4 Summary

In this chapter we saw different ways of scoring classifiers, covered model selection with *cross-validation* and also saw how to use *cross-validation* to find the best regularization parameter.

5.5 Further Reading

1. Alpaydin [2], Section 2.7

Chapter 6

Lazy Learning

Lazy vs Eager learning. K-Nearest Neighbours classification and regression. Kernel Density Estimation. Kernel Regression.

6.1 Lazy and Eager Learning

So far, we have approached all machine learning problems with the intent of finding a function that can predict the class or value of new points. This is called *Eager Learning*, a process in which the training data is used to fit some model and form a hypothesis that generalizes how the input features relate to the value to predict. In *Lazy Learning*, this step is delayed until the moment the system is queried. In this chapter we will see some examples of this approach.

6.2 Classification with K-Nearest Neighbours

A k-nearest neighbours classifier is an example of a *lazy learning* method. More specifically, an *instance learning* method, because the algorithm involves comparing new instances with instances in the training set. Keeping the labelled training set, the k-NN classifier will label a new point with the label of the majority of the k points in the training set that are closer to the new point. Figure 6.1 shows the tesselations of a 1-NN, 3-NN and 5-NN classifiers. For a 1-NN classifier, the new points are labelled with the label of the closest point in the training set, resulting in a *Voronoy tesselation*. For classifiers with more neighbours, the labels are determined by the majority label of the k nearest neighbours and the tesselation becomes more complex.

In all cases, the decision surface is piecewise linear, composed of the hyperplanes across which the nearest neighbours change. As the number of neighbours used increases, the classifier becomes less determined by local conditions. Figure 6.2 shows the decision frontiers for 1-NN, 13-NN and 25-NN classifiers.

To implement a k-NN classifier, we need to start by defining a distance function. For continuous numerical features we can use the *Minkowski distance*, or *p-norm*, which is given by

$$D_{x,x'} = \sqrt[p]{\sum_d |x_d - x'_d|^p}$$

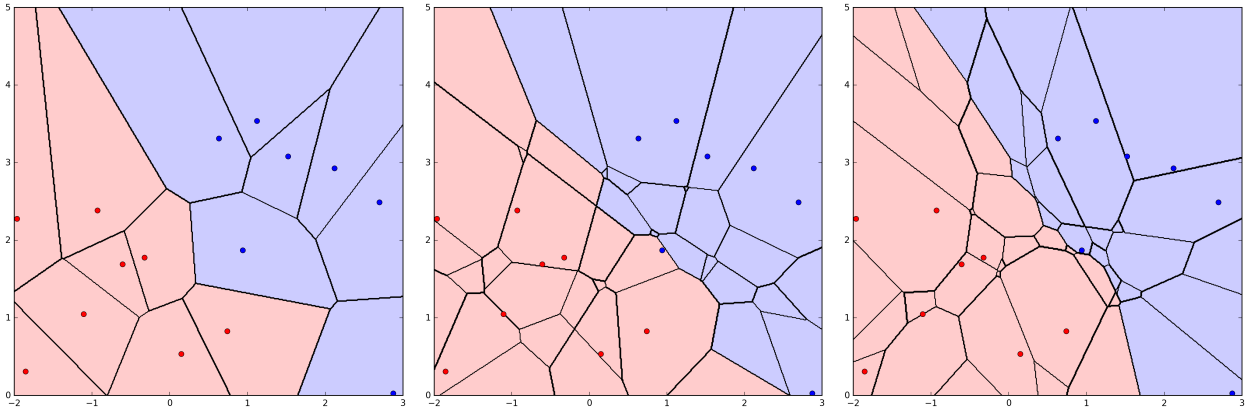

Figure 6.1: Tesselations for 1-NN, 3-NN and 5-NN classifiers.

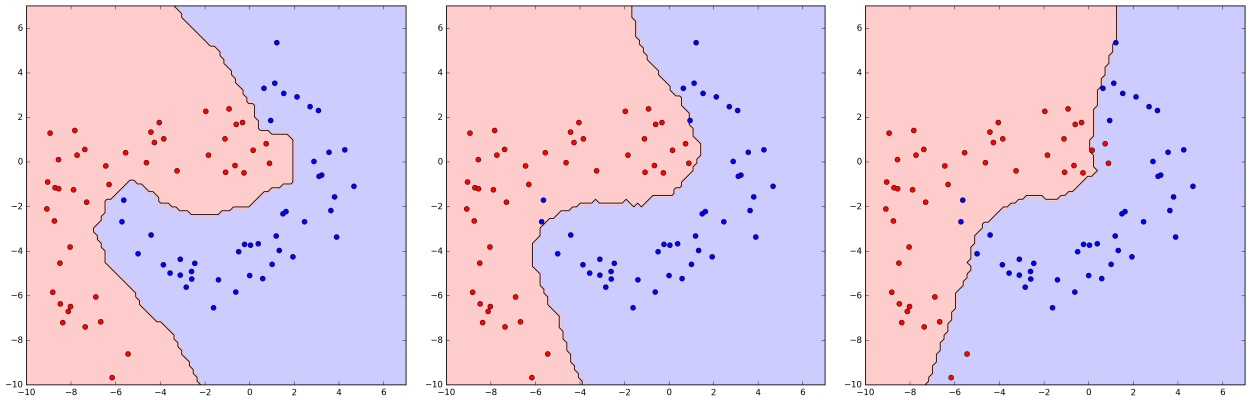

Figure 6.2: Comparison of 1-NN, 13-NN and 25-NN classifiers on the same data set.

Depending on the value of p, this corresponds to the Manhattan distance ($p = 1$) or Euclidean distance ($p = 2$). Other values of p result in different distance measures. For example, for p values between 0 and 1, similarities in one feature become more important. Figure 6.3 shows the effect of different p values.

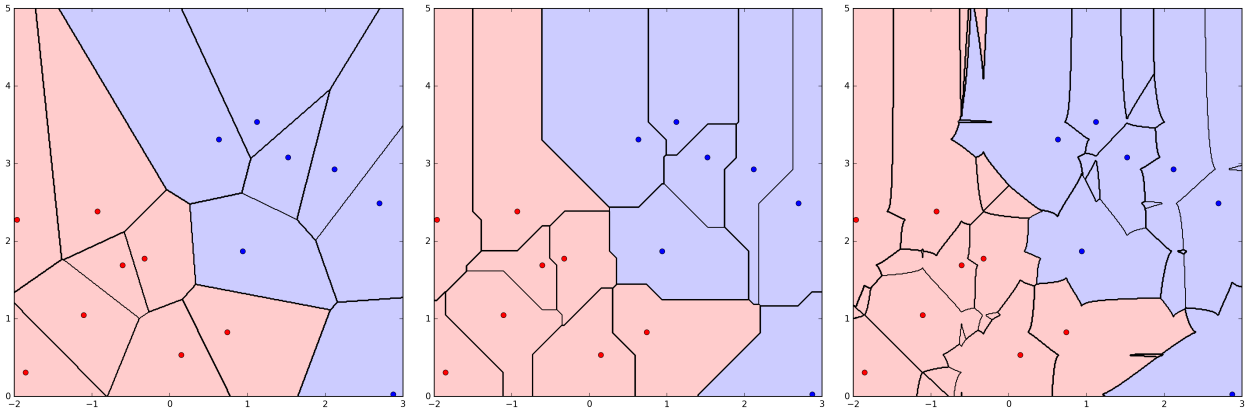

Figure 6.3: Comparison of three distance measures, $p = 2$, $p = 1$ and $p = 0.7$.

For categorical features, a good distance function is the *Hamming distance*:

$$D_{x,x'} = \sum_d \begin{cases} 1, & x_d \neq x'_d \\ 0, & x_d = x'_d \end{cases}$$

Since we are dealing with continuous numerical features, we can start by defining the Minkowski distance, or p-norm, with a default p value of 2, so that it defaults to the euclidean distance.

```
import numpy as np

def mink_dist(x, X, p = 2):
    """return p-norm values of point x distance to vector X"""
    sq_diff = np.power(np.abs(X - x),p)
    dists = np.power(np.sum(sq_diff,1),1.0/p)
    return dists
```

Now we create a function to list the nearest k neighbours in the training set given some example, and then the mode (the most common value) of the label in these nearest k neighbours. This is all we need to classify new data points given the training set X and respective labels Y.

```
from scipy.stats import mode

def k_nearest_ixs(x, X,k):
    """return indexes of k nearest neighbours
    """
    ixs = np.argsort(mink_dist(x,X))
    return ixs[:k]

def knn_classify(x,X,Y,k):
    """return class of x"""
    ix = k_nearest_ixs(x,X,k)
    return mode(Y[ix,0], axis=None)[0][0]
```

Depending on the data and features we have to deal with, it may be desirable to standardize or normalize the inputs. However, this will influence the distance measured between two points and has to be considered with some care. We should do this preprocessing only if we do not wish for features with a greater range of values to weigh more heavily on the distance function. This may often be the case but there can be exceptions. Suppose, for example, that we are dealing with geographical coordinates and want to predict some property of a point by looking up the properties of the neighbours. In that case we should not standardize our data because that would distort the distances by shrinking our data distribution in the direction it spreads the most.

6.3 Example of k-NN Classification

We can use cross-validation to determine the best k value. We load the data set, set aside a third of the points for testing, and then plot the training and validation error with 10-fold cross-validation. Figure 6.4 shows this process. The first panel shows the data set, the second panel the plot of the errors as a function of the k value and the third panel the best model obtained by cross-validation, with $k = 9$. Note that we also plotted the test error. However, we cannot use the test error to choose the model, otherwise the test error would no longer be an unbiased estimator of the true error.

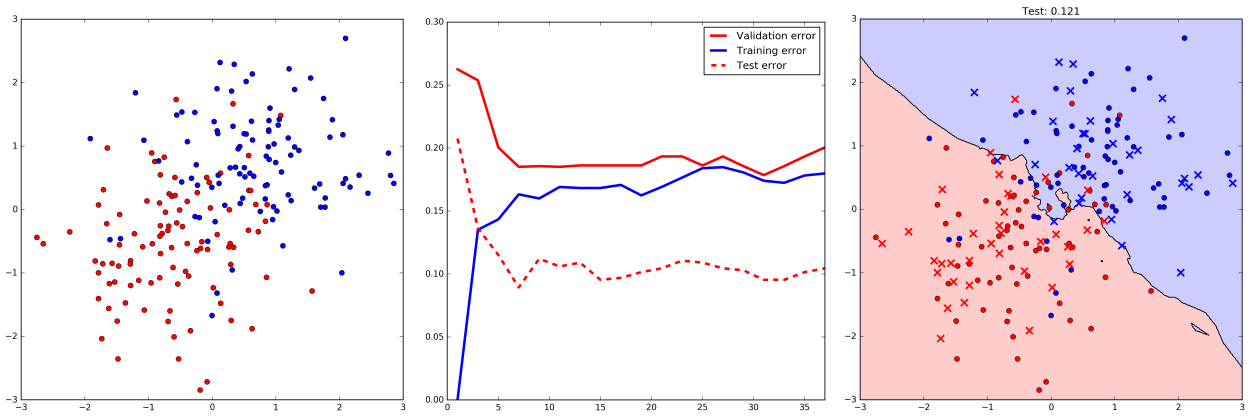

Figure 6.4: Finding the best k value for the k-NN classifier. The first panel shows the data, the second the training, validation and test error plot, and the last panel shows the result, with $k = 9$.

6.4 Curse of Dimensionality

The *curse of dimensionality* is a generic term for a set of problems that arise from dealing with data with many dimensions. In the case of distance-based methods, the problem of high dimensionality is that, with many dimensions, most points are at the frontier of any region. Figure 6.5 shows the proportion of an N-dimensional sphere occupied by another sphere whose diameter is 95

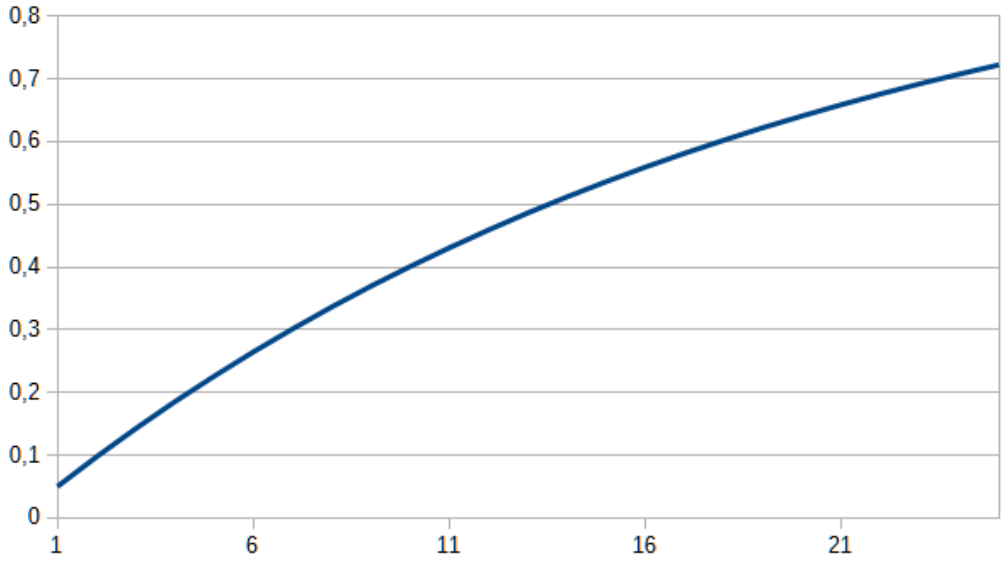

Figure 6.5: Fraction of region occupied by a frontier that is 5% of the diameter as a function of dimension.

6.5 Instance Based Regression

The k-NN approach can also be used for regression, making the predicted value equal to the average of the values of the k nearest neighbours. Figure 6.6 shows a regression curve using different values of k.

However, a better way to perform instance based regression is to use a continuous function that reduces the weight of points farther from the point of interest, and then estimate the desired value using a weighted average of these values. This is called *kernel regression*. The function that weighs

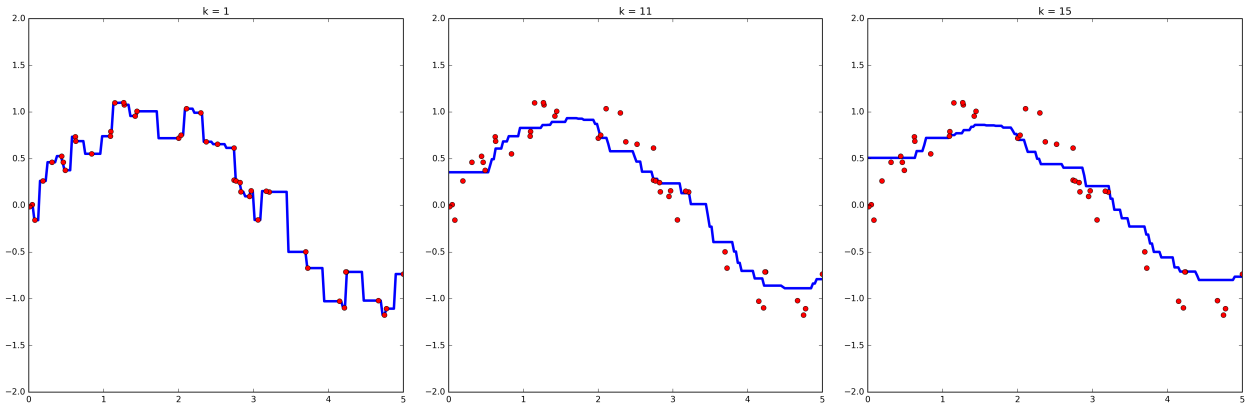

Figure 6.6: K-Nearest Neighbours regression with different values of k.

different points in the training set according to distance is a *kernel function*. A *kernel function* $K(u)$ is a function that satisfies these three conditions:

$$K(u) \geq 0 \ \ \forall u \tag{6.1}$$

$$\int_{-\infty}^{\infty} K(u)du = 1 \tag{6.2}$$

$$K(-u) = K(u) \ \ \forall u \tag{6.3}$$

An example of an often used *kernel function* is the *gaussian kernel*[1].

$$K(u) = \frac{1}{\sqrt{2\pi}} e^{-\frac{u^2}{2}}$$

Then we also need an estimator that predicts the value at x from some function of the y values in the data set weighted by the *kernel function*. For example, the *Nadaraya-Watson* estimator:

$$\hat{y}(x) = \frac{\sum_{t=1}^{N} K\left(\frac{x-x^t}{h}\right) y^t}{\sum_{t=1}^{N} K\left(\frac{x-x^t}{h}\right)}$$

or the *Priestley-Chao* estimator

$$\hat{y}(x) = \frac{1}{h} \sum_{t=2}^{N} (x^t - x^{t-1}) K\left(\frac{x - x^t}{h}\right) y^t$$

For example, we can implement the *gaussian kernel* and the *Nadaraya-Watson* estimator:

```
1 def gaussiank(u):
2   k=np.e**(-0.5*u**2)/np.sqrt(2*np.pi)
3   return k
4
5 def nad_wat(K, h, X, Y, x):
6     num = 0
7     den = 0
8     for ix in range(len(X)):
```

[1] A list of common kernel functions can be found on Wikipedia: https://en.wikipedia.org/wiki/Kernel_(statistics)

```
9           yf = Y[ix]
10          u = (x-X[ix])/h
11          k = K(u)
12          den = den + k
13          num = num + yf * k
14      return num/den
```

And then use them to compute the regression curve from the data, as shown in Figure 6.7.

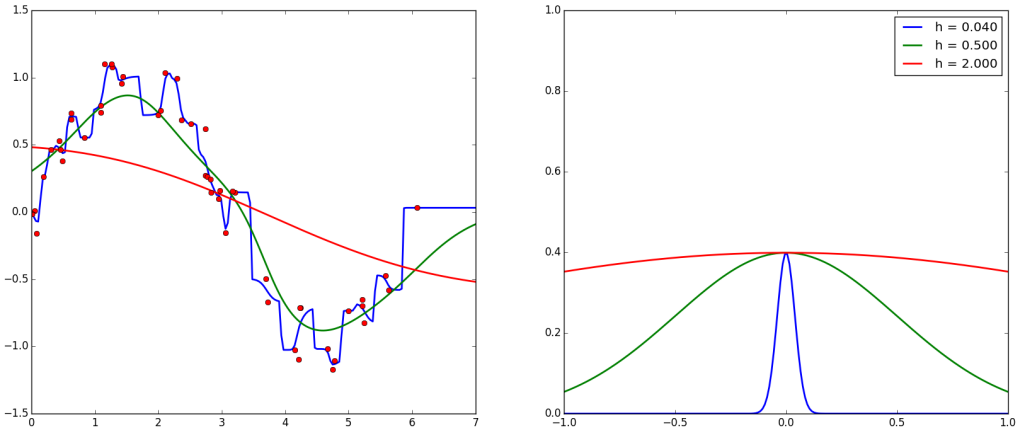

Figure 6.7: Kernel regression with a *gaussian kernel* and a *Nadaraya-Watson* estimator. The three lines show the effect of three different values of the parameter h.

6.6 Kernel Density Estimation

Kernel functions can also be used to smooth density estimates. Given a distribution of points, for example sampled from a normal distribution as shown in the left panel of Figure 6.8, we can depict the varying density of the points using histograms. However, histograms are discontinuous and the result is very dependent on bin size. An alternative is to apply a kernel function to each point and then sum them all. This, shown on the right panel, and leads to a much smoother estimate.

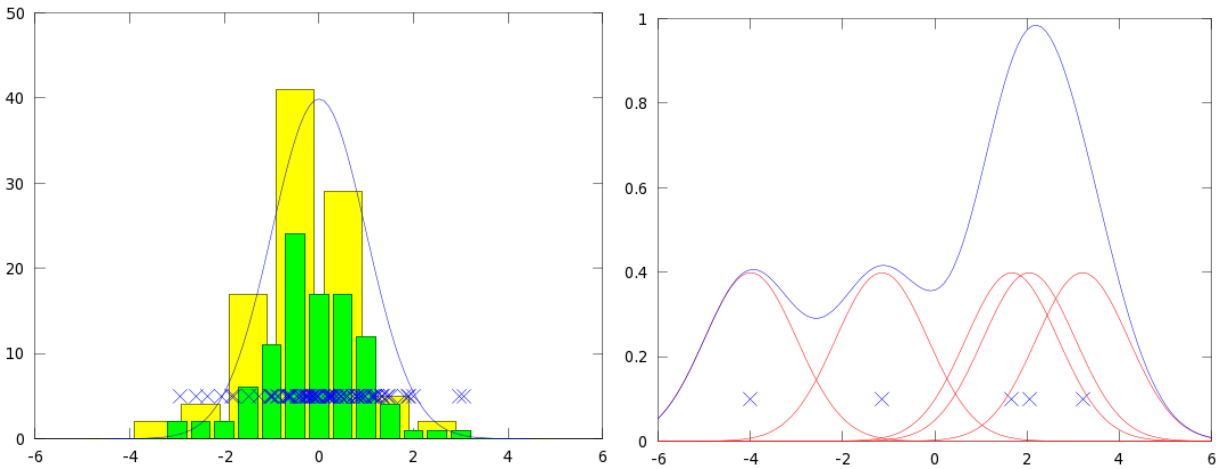

Figure 6.8: Kernel density estimation.

6.7 Summary

In this chapter we saw *lazy learning*, in which inference from the data is delayed until the moment the system is queried, in contrast with *eager learning*, which we covered before, and which involves first training a model of the data. K-nearest neighbours is a *lazy learning* technique for predicting values of new points based on the neighbouring points of the training data, for some distance function. Finally, we covered kernel regression and density estimation.

6.8 Further Reading

1. Alpaydin [2], Sections 8.1 through 8.4
2. Mitchell [18], Sections 8.1 and 8.2
3. Marsland [17], Section 8.4.

Chapter 7

Naïve Bayes

Bayes Classifier and Naïve Bayes Classifier. Parametric and non-parametric models. Generative vs Discriminative classifiers. Comparing classifiers.

7.1 Bayes rule

Let us imagine we have two random variables, X and Y. The probability of $X = x_i$ and $Y = y_j$ is called the *joint probability* and is represented as:

$$p(X = x_i, Y = y_j)$$

The probability of $X = x_i$ is the sum of the joint probabilities of all Y values and $X = x_i$:

$$p(X = x_i) = \sum_{j=1}^{N} p(X = x_i, Y = y_j)$$

This is called the *sum rule* of probability. If we imagine representing the possible values of X and Y in a matrix counting the probability of each combination, $p(X = x_i)$ is obtained by summing the respective column. This is called the *marginal probability* because we can imagine summing it on the margin of the matrix with the *joint probabilities*, as shown in Table 7.1.

The *conditional probability* for $Y = y_j$ given that $X = x_i$, written $p(Y = y_j|X = x_i)$, is the proportion of $p(X = x_i, Y = y_j)$ to $p(X = x_i)$:

$$p(Y = y_j|X = x_i) = \frac{p(X = x_i, Y = y_j)}{p(X = x_i)}$$

This means that

$$p(X = x_i, Y = y_j) = p(Y = y_j|X = x_i)p(X = x_i)$$

This is the *product rule*, which relates the joint probability distribution to the conditional and marginal probabilities. More briefly, these rules can be summarized as follows:

$$\begin{aligned} \textit{sum rule} \qquad & p(X) = \sum_{j=1}^{N} p(X, Y_j) \\ \textit{product rule} \qquad & p(X, Y) = p(Y|X)p(X) \end{aligned}$$

Table 7.1: Joint and Marginal probabilities

Y	X	1	2	3	4	P(Y)
2		0,06	0,026	0,051	0,012	0,189
3		0,045	0,001	0,046	0,016	0,152
4		0,035	0,015	0,065	0,045	0,218
5		0,006	0,033	0,057	0,039	0,157
6		0,029	0,004	0,054	0,035	0,127
P(X)		0,175	0,079	0,273	0,147	

This table shows the joint probabilities for different combinations of X and Y. The marginal probabilities are computed on the margins by summing the respective rows and columns.

Given that joint distributions are symmetric, $p(Y, X) = p(X, Y)$ (just transpose the matrix on Table 7.1), applying the *product rule* we can obtain *Bayes' rule*:

$$p(Y, X) = p(X, Y) \Leftrightarrow p(Y|X)p(X) = p(X|Y)p(Y) \Leftrightarrow p(Y|X) = \frac{p(Y)p(X|Y)}{p(X)}$$

A *frequentist* interpretation will see these probabilities as the frequency of random events in the limit of an infinite number of trials. For example, saying that a coin has a 50% probability of falling "tails" means that, as the number of trials grows to infinity, the fraction of "tails" will tend towards 0.5. But a Bayesian interpretation of probabilities sees the probability values as a measure of our knowledge about the propositions. Under this interpretation, we can see *Bayes' rule* as telling us that the probability of hypothesis Y being true (i.e. our knowledge of Y) given evidence X, which is $p(Y|X)$, has been modified relative to the prior probability of Y, which is $p(Y)$, by the probability of X given Y, or the *likelihood* of Y, written $p(X|Y)$, normalized by the probability of the data X.

This interpretation allows us to consider the probability of an example x belonging to class c as the conditional probability of class C given the features of x: $p(C = c|X = x)$, which would not make as much sense in a frequentist interpretation, unless we assumed the class was determined by the features only with some probability.

7.2 Bayes Classifier

Using *Bayes' rule*, we can write that the probability of an example with feature vector x belonging to class c is:

$$p(C = c|X = x) = \frac{p(C = c)p(X = x|C = c)}{p(X = x)}$$

In other words, the probability of x belonging to c is the prior probability of any point belonging to c multiplied by the likelihood of $C = c$ and divided by the probability of drawing example x at random. Since the probability of drawing example x does not depend on our classifier, we can simplify this expression to:

$$p(C = c|X = x) \propto p(C = c)p(X = x|C = c)$$

But we know from the *product rule* that $p(C = c)p(X = x|C = c)$ is the joint distribution $p(C = c, X = x)$. So if we can compute the joint distribution of the classes and examples, we can choose the

best class for each example. This is the *Bayes classifier*:

$$C^{Bayes} = \underset{c \in \{0,1,...,N\}}{\operatorname{argmax}} p(C = c, X = x)$$

The *Bayes classifier* is ideal in the sense that it minimizes the probability of misclassifying an example. However, it is generally not feasible to compute the joint probability of the classes and features. To understand this, imagine we want to predict if a person has diabetes. We start with a sample of healthy and diabetic individuals and have each fill in a questionnaire with 20 questions on exercise practices, food, smoking, other diseases and so on. Even if the questions are only "yes" or "no", 20 questions gives us about a million combinations. To obtain a reasonable estimate of the joint probability distribution of classes (diabetic or healthy) and all combinations of possible answers we would need millions of volunteers and questionnaires. Without simplifying assumptions we cannot do this. In short, although the *Bayes classifier* is the ideal classifier in theory, in practice it is generally impossible to use.

7.3 Naïve Bayes Classifier

In the previous section, we saw that we can predict the class of an example by finding the maximum of the joint probability of each class and the features of that example. We can decompose this using the *product rule* as follows, considering $x_1, ..., x_n$ to be the components of the feature vector and C_k the probability of the example being in class k:

$$p(C_k, x_1, x_2, ..., x_n) = p(C_k)p(x_1|C_k)p(x_2|C_k, x_1)...p(x_n|C_k, x_1, x_2, ..., x_{n-1})$$

Variables A, B are *conditionally independent* given X if:

$$p(A, B|X) = P(A|X)P(B|X)$$

That is, if their joint probability conditioned on the other variable is just the product of their probabilities conditioned on that other variable.

An example of conditional independence could be the time two persons living in the same neighbourhood arrive at home from work. These variables may not be independent because, whenever there is a strike in the public transport system, both will arrive later. So if one arrives late it is more likely that the other arrived late too. However, if we know that there was such a strike, then knowing when one of them arrived home gives us no new information about when the other will arrive, and thus the two are independent if we know if there was or was not a strike.

So, if we assume that the feature values $x_1, ..., x_n$ are *conditionally independent* given the class, it follows that:

$$p(x_n|C_k, x_1, x_2, ..., x_{n-1}) = p(x_n|C_k)$$

for any n. This allows us to greatly simplify the computation of the joint distribution:

$$p(C_k, x_1, x_2, ..., x_n) = p(C_k) \prod_{j=1}^{N} p(x_j|C_k)$$

or, if we take the logarithms to prevent numeric overflow or underflow problems:

$$\ln p(C_k, x_1, x_2, ..., x_n) = \ln p(C_k) + \sum_{j=1}^{N} \ln p(x_j|C_k)$$

This means that our classifier can be:

$$C^{\text{Naïve Bayes}} = \underset{k\in\{0,1,\ldots,K\}}{\operatorname{argmax}} \ln p(C_k) + \sum_{j=1}^{N} \ln p(x_j|C_k)$$

This is called the *Naïve Bayes classifier* because of the assumption that all features are *conditionally independent* on the class. In general, this is not true. However, since we are not concerned with the absolute probability values but merely with finding the class that maximizes these values, the *Naïve Bayes classifier* tends to work rather well. In addition, it is very easy to apply. If we consider again the diabetes example of the previous section, for a *Naïve Bayes classifier* we would only need to find the probability distribution of each feature given the class. So we would only need to compute the proportions of yes and no for each answer in all questionnaires given to healthy subjects and the same for all questionnaires given to diabetic subjects. This should easily be done with a few dozen questionnaires instead of millions.

7.4 Naïve Bayes, example 1: continuous features

Let us consider a data set where each point has two continuous features and belongs to one of two classes. To train a *Naïve Bayes classifier*, we need to determine the conditional probability distribution of each feature given each class. With features that have continuous values we have several options. One is to use a *parametric model*. For example, if we assume that a feature is a normally distributed random variable when conditioned on the class, we can compute its probability distribution using the normal distribution:

$$p(x_j|C_k) = \frac{1}{\sigma_k\sqrt{2\pi}} e^{-\frac{(x-\mu_k)^2}{2\sigma_k^2}}$$

where μ_k and σ_k are, respectively, the mean and standard deviation of the values of feature x_j for all points in class C_k. This is a *parametric model* because the model is completely determined by a specific set of parameters, and there are different probability distributions that we can consider. Alternatively, we can use a *nonparametric model* for the distribution. This is a model that, even though it can have parameters, it is not completely determined by the parameters. A histogram is an example of a *nonparametric model*. It has one parameter – the size of the bins used to partition the values – but it cannot be completely determined by that parameter, since we also need to count the values. Another example of a *nonparametric model* for these distributions is a *Kernel Density Estimator*, as we saw in Section 6.6. Figure 7.1 compares these three models for finding the distribution of one feature from one of the classes of our data set.

A kernel density estimator seems to be the best option, and it generally is unless we know the distribution function and can use a parametric model. So now we load the data and find the distributions for each of the two features in each of the two classes. The product of these distributions, for each class, is our estimate of the joint probability distribution under the naïve assumption that the features are conditionally independent given the class, which is the assumption used in the *Naïve Bayes classifier*. Figure 7.2 shows the data, the four KDE computed (two classes times two features) and the 3D plot showing the products of the probability distributions for each class, which, under the assumption of conditional independence, are the estimates of the joint probability distributions of the features given each class. The KDE was computed using a gaussian kernel and the Nadaraya-Watson estimator, as illustrated in Section 6.6.

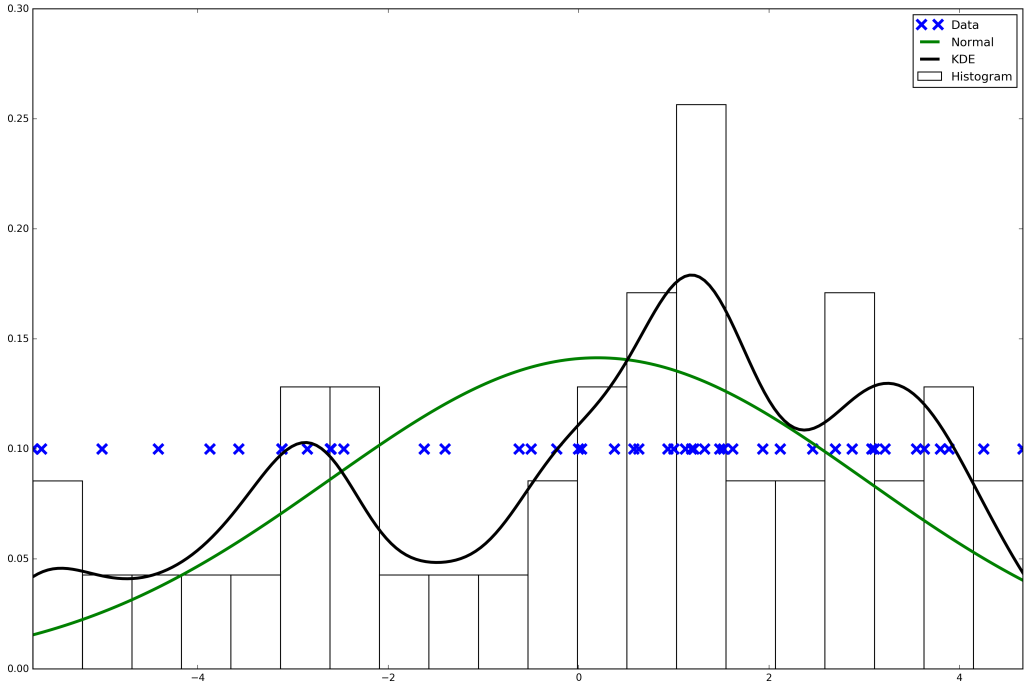

Figure 7.1: Different distribution estimates for one feature in one class of our data set. The data values are in one dimension, and represented with Y = 0.1 just to make it easier to see them.

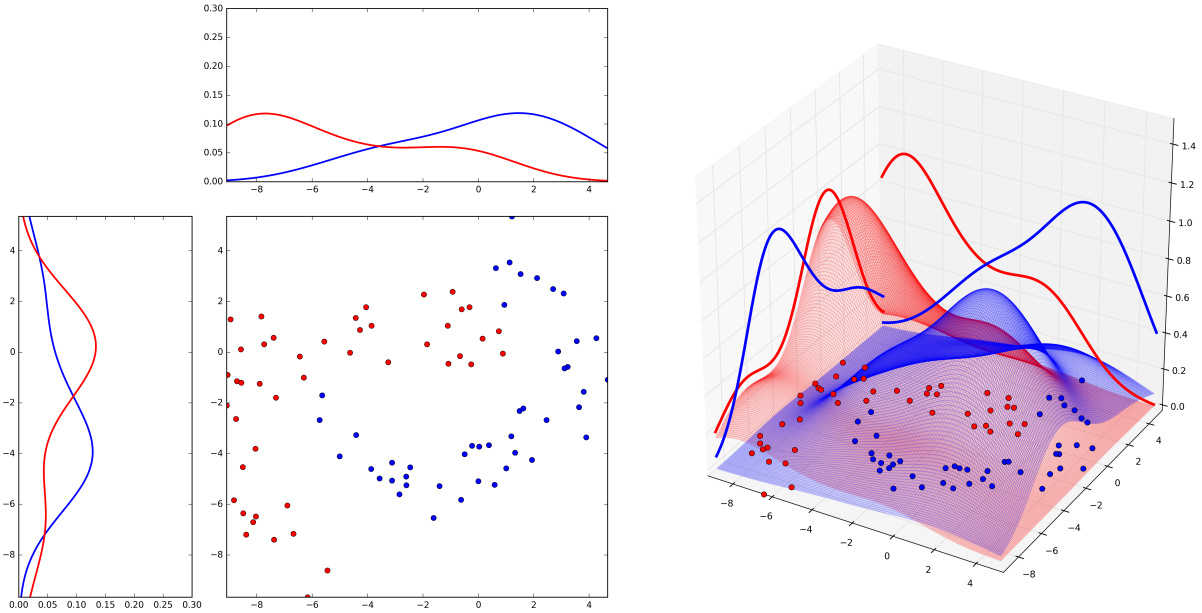

Figure 7.2: Kernel density estimation of the four distributions and the estimated joint distributions conditioned on the class (red or blue) under the Naïve Bayes assumption that the features are conditionally independent given the class. Note that the Z scale in the vertical plot was normalized to a maximum of 1 so the shape of the product plots are easier to see.

Now we just need to consider the proportion of red and blue class points in our data (the $p(C_k)$ term and find, for each point to classify, the class that maximizes:

$$C^{\text{Naïve Bayes}} = \underset{k \in \{0,1,\ldots,K\}}{\operatorname{argmax}} \ln p(C_k) + \sum_{j=1}^{N} \ln p(x_j | C_k)$$

However, the KDE we used has one parameter h, which determines the width of the kernel function,

and different values of h lead to different classifiers. Figure 7.3 shows the result of the classifier with different values of h.

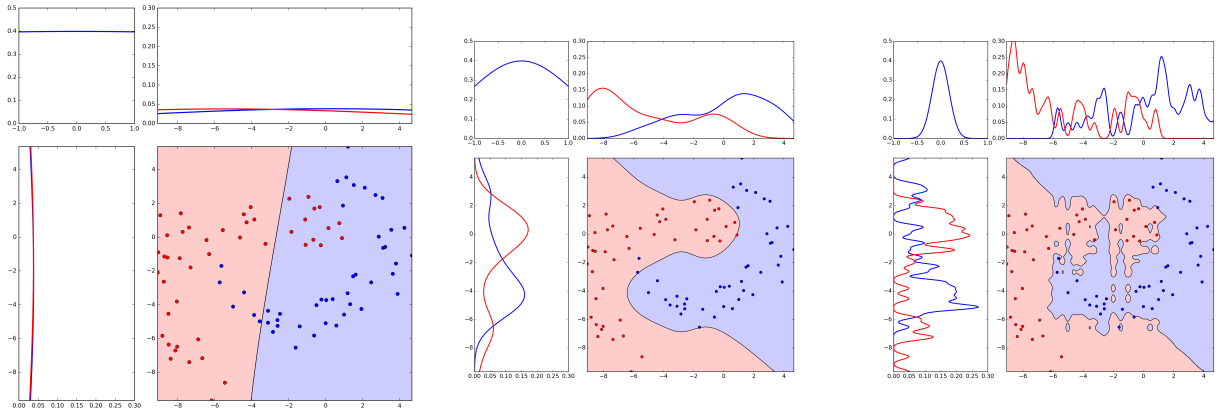

Figure 7.3: The Naïve Bayes classifier trained with this data set using different values of h for the KDE. In each panel, the top-left plot depicts the kernel function resulting from the respective h value.

To determine the best value we can use *cross-validation*. Figure 7.4 shows the result of 10-fold cross validation, depicting the training and validation errors as a function of the value of h. The best value, minimizing the validation error, was $h = 1.8$. The right panel shows the classifier retrained with the complete training set and using $h = 1.8$ for the kernel density estimators.

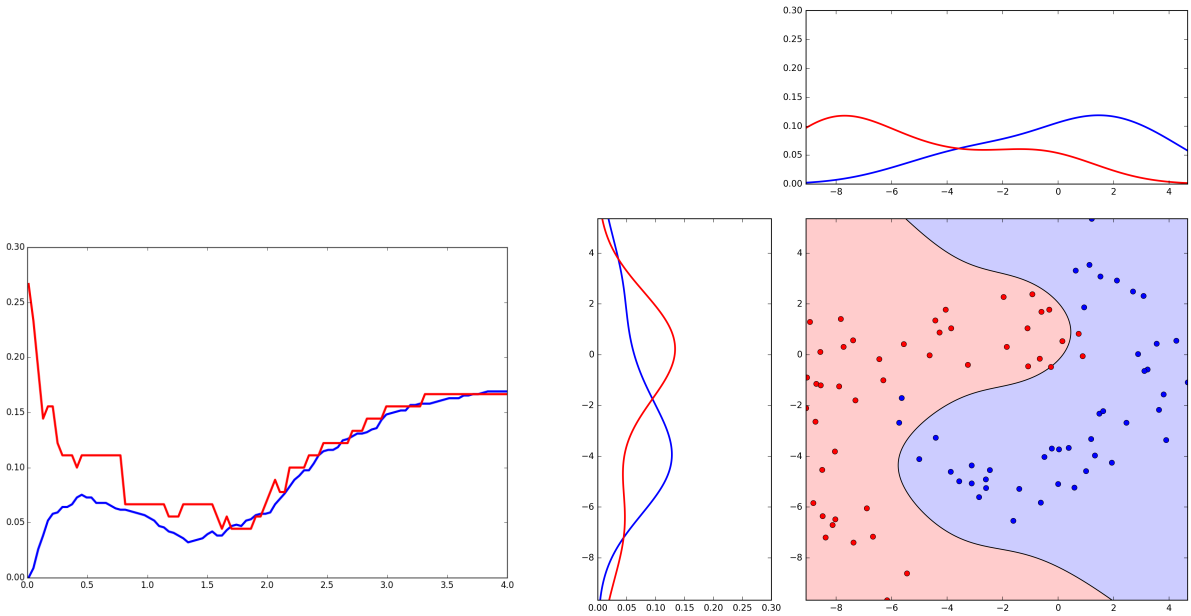

Figure 7.4: Cross-validation results and the final Naïve Bayes classifier for $h = 1.8$.

7.5 Naïve Bayes, example 2: categorical featues

For this example, we will be using a data set describing mushroom samples with 22 categorical features, each labelled as edible or poisonous[1]. We will be using a *Naïve Bayes classifier* to try to predict if a mushroom is edible. The features are all categorical and described in a features file:

[1]From the UCI machine learning repository: http://archive.ics.uci.edu/ml/datasets/Mushroom

```
1. cap-shape: bell=b,conical=c,convex=x,flat=f, knobbed=k,sunken=s
2. cap-surface: fibrous=f,grooves=g,scaly=y,smooth=s
[...]
21. population: abundant=a,clustered=c,numerous=n, scattered=s,several=v,solitary=y
22. habitat: grasses=g,leaves=l,meadows=m,paths=p, urban=u,waste=w,woods=d
```

The data is stored as strings with one sample per line. The first character indicates the class, with `p` for poisonous and `e` for edible. The rest of the line indicates the value for each feature with the corresponding character codes, separated by commas.

```
p,x,s,n,t,p,f,c,n,k,e,e,s,s,w,w,p,w,o,p,k,s,u
e,x,s,y,t,a,f,c,b,k,e,c,s,s,w,w,p,w,o,p,n,n,g
e,b,s,w,t,l,f,c,b,n,e,c,s,s,w,w,p,w,o,p,n,n,m
p,x,y,w,t,p,f,c,n,n,e,e,s,s,w,w,p,w,o,p,k,s,u
e,x,s,g,f,n,f,w,b,k,t,e,s,s,w,w,p,w,o,e,n,a,g
[...]
```

First we will load the information on the possible values for each feature. We will also add a possible value of "?" because, in some cases, the value is missing and missing values are indicated by this character. This function reads all the lines in the features file (specially modified so that each feature description is in a single line in the text file), splits each line on the = character and stores the following character.

```
1  def get_features():
2      lines = open('agaricus-lepiota.features').readlines()
3      features = []
4      for lin in lines:
5          ft_vals = '?'
6          fragments = lin.split('=')
7          for frag in fragments[1:]:
8              ft_vals = ft_vals+frag[0]
9          features.append(ft_vals)
10     return features
```

With the list of strings describing the possible values for the features, we can now load the data. This function removes the commas separating the attribute values then fills in a matrix with the index of each code. Before returning the features and class matrices, this function also shuffles the ordering of the rows. The purpose of this is to remove any correlations present in the ordering of the data file.

```
1  def load_data(features,class_codes):
2      lines = open('agaricus-lepiota.data').readlines()
3      feat_vals = np.zeros((len(lines),22)).astype(int)  # to store indexes
4      classes = np.zeros(len(lines))
5      for row,lin in enumerate(lines):
6          s = lin.replace(',','').strip()
7          classes[row] = class_codes.index(s[0])
8          for column,fv in enumerate(s[1:]):
9              feat_vals[row,column] = features[column].index(fv)
10     ixs = list(range(feat_vals.shape[0]))
11     np.random.shuffle(ixs)
12     return feat_vals[ixs,:],classes[ixs]
```

Now we can estimate the conditional probability distributions of the values for each feature given the class. The following function receives the feature value matrix and the list of possible codes for each feature. Since the features are all categorical, it is best to use histograms. The only detail to remember here is to avoid having values with a probability of zero. This can happen if the value is absent from the training set. To prevent this, we can use *additive smoothing*. Instead of simply computing the fraction of occurrences of each value, we also add a constant α:

$$\hat{p}(x_j = k) = \frac{count(k) + \alpha}{N + \alpha d}$$

where d is the number of possible values in feature j. This function creates a list of vectors, each vector counting the occurrences of the different possible values of the corresponding feature, starting with 1 as the value of the α constant. After counting, the function computes the logarithm of the fraction for each value. Logarithms are useful in this case so we can sum instead of multiplying the values.

```
def make_hists(data,features):
    hists = []
    for feat in features:
        hists.append(np.ones(len(feat)))
    for row in range(data.shape[0]):
        for column in range(data.shape[1]):
            hists[column][data[row,column]] +=1
    for ix in range(len(hists)):
        hists[ix] = np.log(hists[ix]/float(data.shape[0]+len(features[ix])))
    return hists
```

Now we need to load the data and split it into a training and test set. Previously, we have done this with *random sampling*, which consists of splitting the sets at random. However, it is best to have the same proportion of the two classes in the training and test set. So this time we will use *stratified sampling*. First we split the data in two sets, corresponding to the edible and poisonous examples. Then we draw the same fraction of each set for the test set. Since the `load_data` function shuffles the examples at random, this is easy to do by simply splitting the matrices in two.

```
def split_data(features,test_fraction):
    feat_vals,classes = load_data(features,'ep')
    edible = feat_vals[classes==0,:]
    poison = feat_vals[classes==1,:]
    e_test_points = int(test_fraction*edible.shape[0])
    e_train = edible[e_test_points:,:]
    e_test = edible[:e_test_points,:]
    p_test_points = int(test_fraction*poison.shape[0])
    p_train = poison[p_test_points:,:]
    p_test = poison[:p_test_points,:]
    return e_train,p_train,e_test,p_test
```

Now all we need is a function to classify examples. The function `classify` receives the histograms with the logarithms of the estimated probabilities and the logarithm of the prior probability of an example belonging to either class, $p(C_k)$. This is simply the logarithm of the fraction of each class in the data. This function sums all the terms in this equation:

$$C^{\text{Naïve Bayes}} = \underset{k \in \{0,1,\ldots,K\}}{\operatorname{argmax}} \ln p(C_k) + \sum_{j=1}^{N} \ln p(x_j | C_k)$$

and determines the class according to the maximum value found.

```
def classify(e_class,e_log,p_class,p_log,feat_mat):
    classes = np.zeros(feat_mat.shape[0])
    for row in range(feat_mat.shape[0]):
        e_sum = e_log
        p_sum = p_log
        for column in range(feat_mat.shape[1]):
            e_sum = e_sum + e_class[column][int(feat_mat[row,column])]
            p_sum = p_sum + p_class[column][int(feat_mat[row,column])]
        if e_sum<p_sum:
            classes[row]=1
    return classes
```

Now we put it all together and evaluate the performance of our classifier on the test set by computing the percentage of misclassifications.

```
def do_bayes():
    features = get_features()
    e_train,p_train,e_test,p_test = split_data(features,0.5)
    e_hists = make_hists(e_train,features)
    p_hists = make_hists(p_train,features)
    tot_len = e_train.shape[0]+p_train.shape[0]
    e_log = np.log(float(e_train.shape[0])/tot_len)
    p_log = np.log(float(p_train.shape[0])/tot_len)
    c_e = classify(e_hists,e_log,p_hists,p_log,e_test)
    c_p = classify(e_hists,e_log,p_hists,p_log,p_test)
    errors = sum(c_e)+sum(1-c_p)
    error_perc = float(errors)/(len(c_e)+len(c_p))*100
    print('%d errors;' % errors, ' %.2f%% error rate' % error_perc)
```

We can also look at the *confusion matrix* by counting the correct and incorrect classifications of edible and poisonous mushrooms:

		Real class	
		Edible	Poisonous
Predictions	Edible	2089	221
	Poisonous	15	1737

From the *confusion matrix* we can see that most of the mistakes in classification are in classifying as edible mushrooms that are poisonous. This is a more costly mistake than mistaking edible mushrooms for poisonous ones, and it suggests one problem that we have not considered so far, which is that minimizing misclassification alone is not the ideal option when different errors have different costs.

7.6 Discriminative and Generative classifiers

So far we saw three different classifiers. Logistic regression and k-Nearest Neighbours predict the class of an example from an estimate of the conditional probability of a point belonging to a class given the features. These are examples of *discriminative classifiers*. Naïve Bayes is a *generative classifier*, because in this case the classifier first estimates the joint probability distribution of the classes and features values, and then predicts the class from this joint probability. The reason why this type of

classifier is called *generative* is that the joint probability distribution can be used to generate synthetic examples for each class. Figure 7.5 shows an example of training a *Naïve Bayes classifier* and then using it to generate synthetic data.

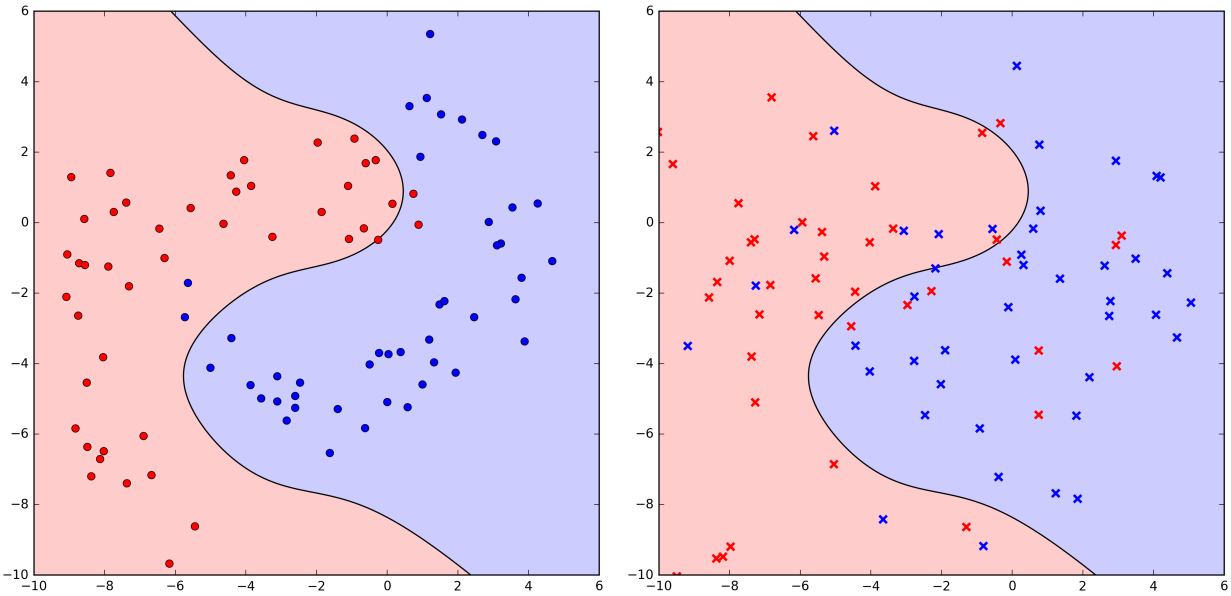

Figure 7.5: Naïve Bayes classifier trained with the data on the left panel, used to generate the set of points on the right panel.

7.7 Comparing classifiers

Figure 7.6 shows three different classifiers trained and tested on the same data: Logistic Regression, k-Nearest Neighbours and Naïve Bayes. These classifiers make, respectively, 10, 6 and 1 misclassification errors on the test set. The question we need to address is whether any of these classifiers is significantly better than the others. One solution is to use an *approximate normal test*. Since the number of errors result from the sum of independent random variables, the number of errors tends towards a normal distribution with a mean equal to the expected number of errors. If the true probability of misclassification is p_0, then the mean will be Np_0 and the standard deviation is $\sqrt{Np_0(1-p_0)}$:

$$\frac{X - Np_0}{\sqrt{Np_0(1-p_0)}} \approx Z$$

where X is the number of misclassified examples and N is the total size of the test set. With this approximation we can estimate a confidence interval for the expected number of errors in the given classifiers, Np_0. For a 95% confidence interval:

$$X - 1.96\sigma < Np_0 < X + 1.96\sigma$$

with $\sigma = \sqrt{Np_0(1-p_0)}$, which we can estimate by estimating $p_0 = X/N$. If the intervals computed for two classifiers do not intersect, we can exclude the hypothesis that they have the same expected error rate p_0. Applying this to our classifiers, we get the following 95% confidence intervals:

$$X^{LogReg} = 10 \pm 5.4 \qquad X^{kNN} = 6 \pm 3.5 \qquad X^{NB} = 1 \pm 1.9$$

This means that we cannot exclude the hypothesis that the first two classifiers have the same true error, since their intervals intersect. Naïve Bayes seems to be a better classifier than Logistic Regression. However, when X is a very small number, this test is not very reliable. As a rule of thumb, X should be above 5, approximately, for this test to be useful.

An alternative method is *McNemar's test*. Let e_{01} be the number of examples the first classifier misclassifies but the second classifies correctly, and e_{10} be the number of examples the second classifier classifies incorrectly but the first classifier classifies correctly. The difference divided by the total follows approximately a chi-squared distribution with one degree of freedom:

$$\frac{(|e_{01} - e_{10}| - 1)^2}{e_{01} + e_{10}} \approx \chi_1^2$$

The -1 term is a continuity correction term because the error counts are discrete and the χ^2 distribution is continuous. If the value is greater than 3.84, we can reject the null hypothesis (that the two classifiers perform identically) with 95% confidence. In our case, the results are:

$$LogReg\ vs\ kNN = 0.8 \qquad kNN\ vs\ NB = 2.3 \qquad NB\ vs\ LogReg = 7.1$$

This means we can conclude there is likely to be a difference between the performance of the Naïve Bayes and the Logistic Regression classifiers, but that the difference is not significant in the other cases.

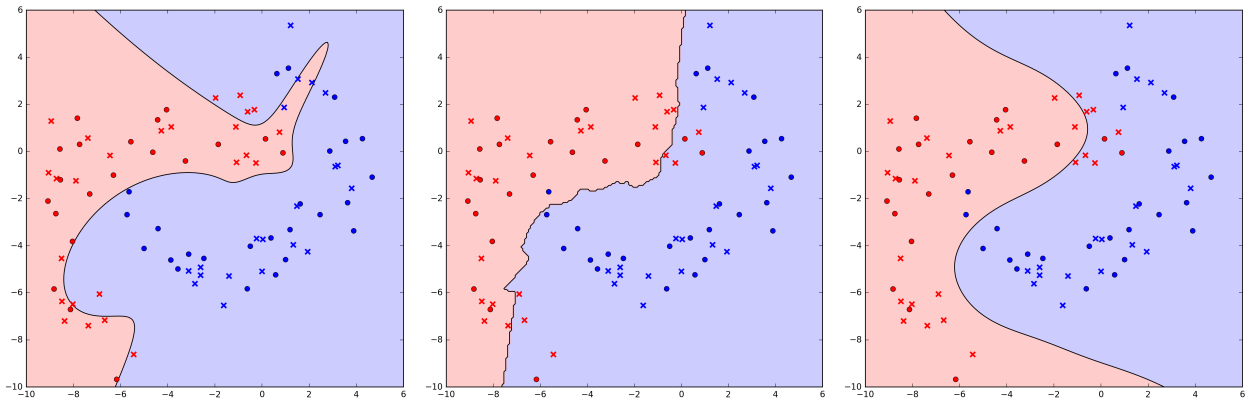

Figure 7.6: Three classifiers: logistic regression, k-NN and Naïve Bayes. All were trained on the set marked as circles and tested on the points marked as crosses.

7.8 Further Reading

1. Bishop [4], Section 1.2
2. Alpaydin [2], Section 14.6
3. Mitchell [18], Section 6.9
4. Marsland [17], Section 8.1.2

Chapter 8

Multi-layer Perceptron

Perceptron. Multi-layer Perceptron. Backpropagation. Regularization in MLP.

8.1 Perceptron

Figure 8.1 shows a neuron cell. Neurons have a set of dendritic branches which can be stimulated by other cells. If the stimulus passes a threshold, then the neuron fires an impulse over the axon, consisting of a wave of membrane depolarization. This in turn leads to the release of neurotransmitters in the synaptic terminals. The neuron provides the inspiration for the *perceptron*. Originally, the *perceptron* model consisted of a linear combination of the inputs, plus a bias value, and a non-linear threshold response function:

$$y = \sum_{j=1}^{d} w_j x_j + w_0 \qquad s(y) = \begin{cases} 1, & y > 0 \\ 0, & y \leq 0 \end{cases}$$

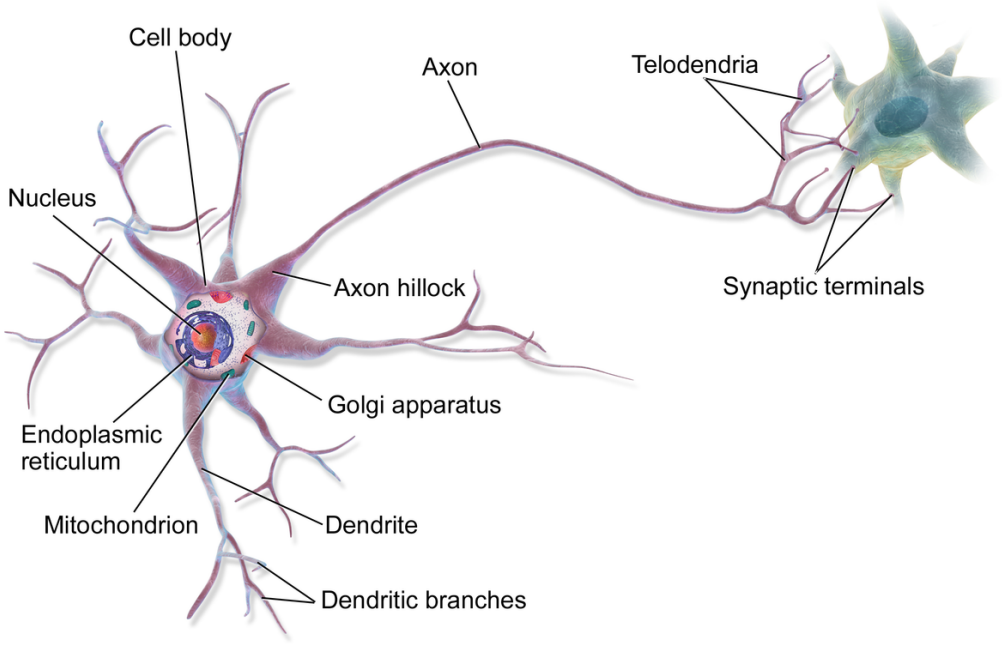

Figure 8.1: Neuron anatomy (BruceBlaus, CC-BY, source Wikipedia).

Note that, as we did in the case of logistic regression, we can include this bias value in the product of the inputs and the coefficients by adding a bias value of 1 to the input vector. The *perceptron* represents a hyperplane that separates the inputs into two classes, 0 and 1. To train a *perceptron*, we present labelled examples and adjust the weights according to the following rule:

$$w_i = w_i + \Delta w_i \qquad \Delta w_i = \eta(t - o)x_i$$

where t is the target label of the example, o the output of the *perceptron* for that example, x_i the input value for feature i and w_i the coefficient i of the perceptron. Since the output of the *perceptron* is either 0 or 1, as is the target class of each example, the training rule consists essentially of adjusting the weights of the *perceptron* for every example that is incorrectly classified. The problem with this original formulation of the perceptron is that the response function is discontinuous. This may be nearer to the biological features of the neuron but raises problems with the minimization of the error functions. An alternative is to use a differentiable threshold function. One often used function is the *logistic* function, also called the *sigmoid* function:

$$s(y) = \frac{1}{1 + e^{-y}} = \frac{1}{1 + e^{-\vec{w}^T \vec{x}}}$$

There are other functions that can be used in this role, such as the *hyperbolic tangent*, for example. However, here we will only focus on the familiar logistic function. Although this is strictly not the same as the perceptron, in the original formulation, it is also common to call this variant a perceptron too.

8.2 A Single Neuron

Training a logistic response neuron can be done by minimizing the squared error between the response of the perceptron and the target class. This is the idea behind the *Brier score* we saw in Chapter 5. So we minimize the error function:

$$E = \frac{1}{2}\sum_{j=1}^{N}(t^j - s^j)^2$$

But we can do this in a way similar to the one used for the *perceptron*, by adjusting the weights of the neuron in small steps as a function of the error at each example j, $E^t = \frac{1}{2}(t^j - s^j)^2$, where t^j is the class of example j and s^j is the neuron's response for example j. To do this, we need to compute the derivative of the error as a function of the weights of the neuron in order to compute how to update the neuron weights. Since the error is a function of the activation of the neuron for example j (s^j), the activation is a function of the weighted sum of the inputs (net^j) and this is, in turn, a function of the weights, we use the *chain rule* for the derivative of compositions of functions to obtain the gradient as a function of each weight:

$$-\frac{\delta E^j}{\delta w} = -\frac{\delta E^j}{\delta s^j}\frac{\delta s^j}{\delta net^j}\frac{\delta net^j}{\delta w}$$

where

$$s^t = \frac{1}{1 + e^{-net^j}} \qquad net^j = w_0 + \sum_{i=1}^{M} w_i x_i$$

Since

$$\frac{\delta net^j}{\delta w} = x$$
$$\frac{\delta s^j}{\delta net^j} = s^j(1 - s^j)$$
$$\frac{\delta E^j}{\delta s^j} = -(t^j - s^j)$$

We obtain the following update rule for the weight i of the neuron given example j:

$$\Delta w_i^j = -\eta \frac{\delta E^j}{\delta w_i} = \eta(t^j - s^j)s^j(1 - s^j)x_i^j$$

Using this update function we descend the error surface in small steps in different directions according to each example presented to the net. With examples presented in random order, this is a *stochastic gradient descent*. Figure 8.2 illustrates this process of stochastically descending the error surface. The process of updating the weights at each example is called *online learning*. An alternative training schedule consists of summing the Δw_i^j updates for the whole training set (an epoch) and then updating the weights with the total. This is called *batch learning*. These are examples of *stochastic gradient descent* because they are ways of descending along the gradient of the error function along random paths depending on the data.

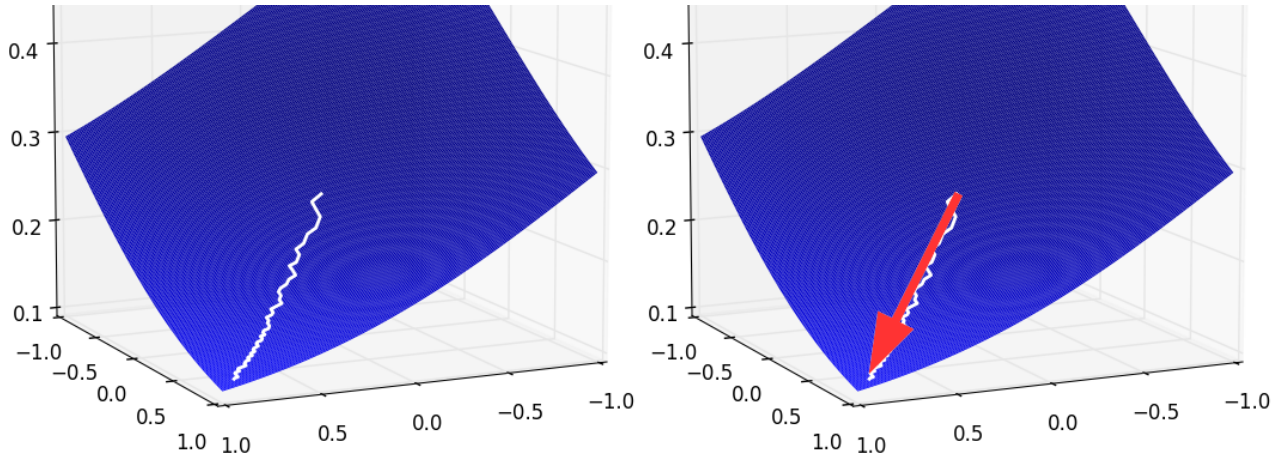

Figure 8.2: Stochastic gradient descent with online training (left panel) and batch training (right panel).

With a single neuron it is possible to learn to classify any linearly separable set of classes. One classical example is the `OR` function, as shown in Table **??**.

Table 8.1: The OR function

x_1	x_2	OR
0	0	0
0	1	1
1	0	1
1	1	1

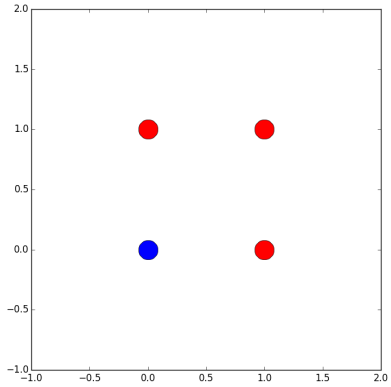

Figure 8.3: Set of points from the OR function.

Figureřef8-neuro-or shows the training error for one neuron being presented the four examples of the OR function and the final classifier, separating the two classes. The frontier corresponds to the line where the response of the neuron is 0.5.

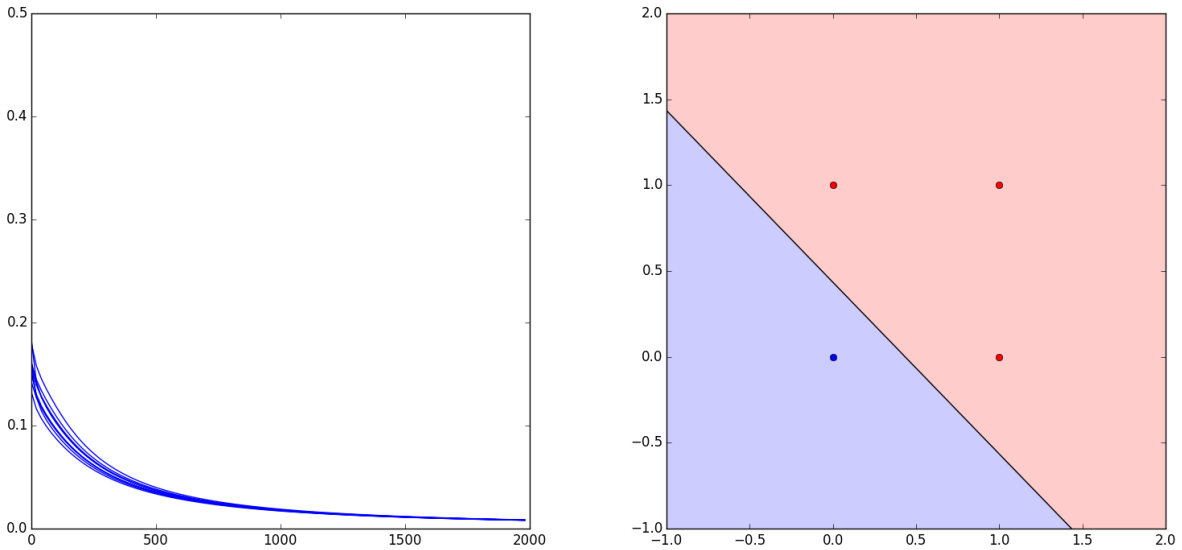

Figure 8.4: Training error and final classifier for one neuron trained to separate the classes in the OR function.

However, if the sets are not linearly separable, a single neuron cannot be trained to classify them correctly. This is because the neuron defines a hyperplane separating the two classes. For example, the exclusive or (XOR) function results in two classes that are not linearly separable, as Table 8.2 illustrates. So, if we try to train a neuron to separate these classes there is no reduction in the training error nor does the final classifier manage to separate the classes, as shown in Figure 8.6.

Table 8.2: The XOR function

x_1	x_2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

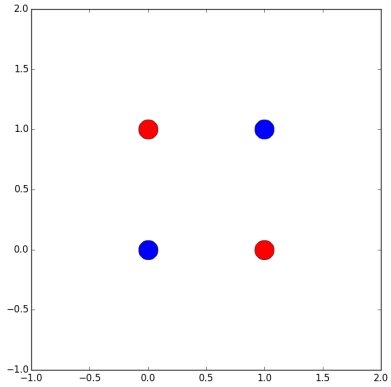

Figure 8.5: Set of points from the OR function.

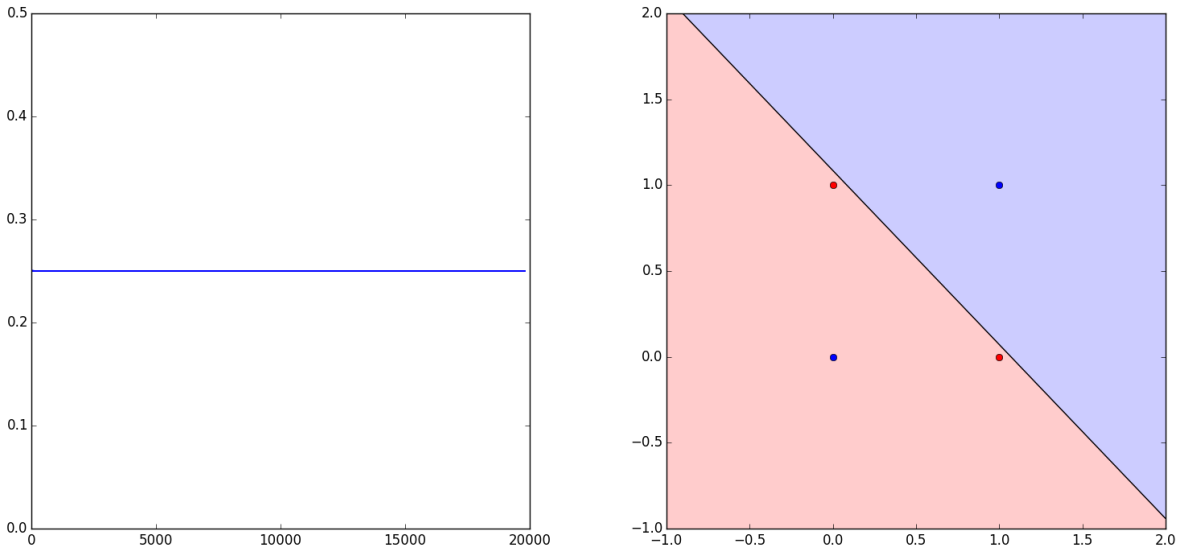

Figure 8.6: Training error and final classifier for one neuron trained to separate the classes in the OR function.

The solution for this problem is to add more neurons in sequence.

8.3 Multilayer Perceptron

The *multilayer perceptron* is a fully connected, feedforward neural network. This means that each neuron of one layer receives as input the output of all neurons of the layer immediately before. Figure 8.7 shows two examples of multilayer perceptrons (MLP).

To update the coefficients of the output neurons, we derive the same update rule as for the single neuron with the only difference that the input value is not the value of an example feature but rather the value of the output of the neuron from the previous layer. Thus, the update rule for weight m of neuron n in layer k is:

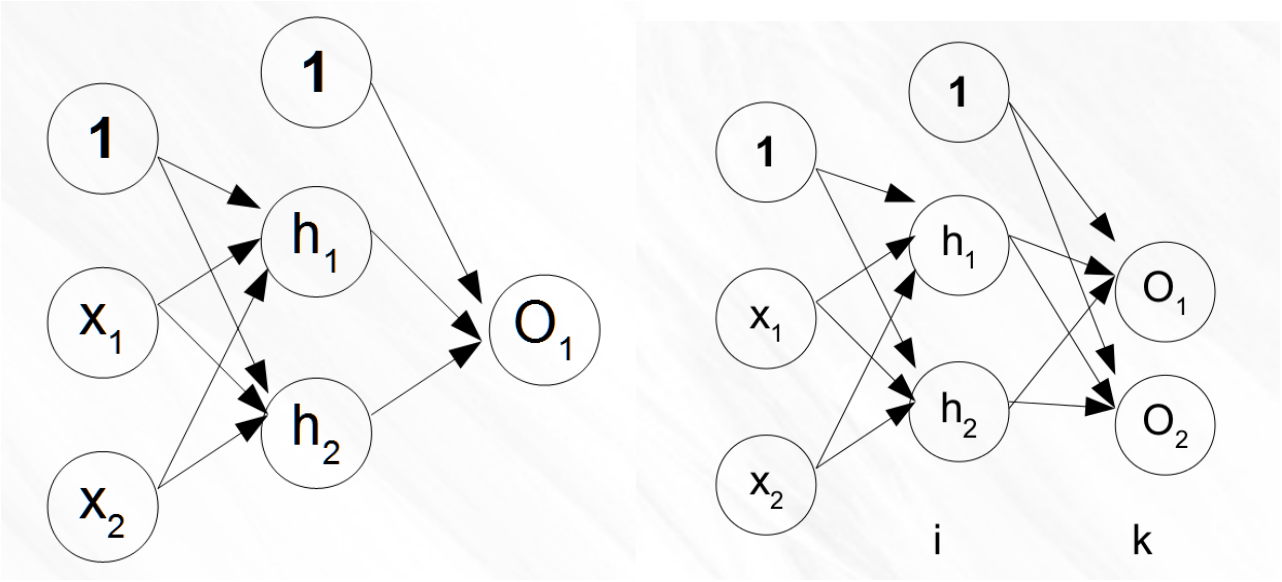

Figure 8.7: Two examples of multilayer perceptrons. Both have a hidden layer. The left panel shows a MLP with one output neuron, the right panel an MLP with two output neurons.

$$\begin{aligned}\Delta w^j_{m,k,n} &= -\eta \frac{\delta E^j_{k,n}}{\delta s^j_{k,n}} \frac{\delta s^j_{k,n}}{\delta net^j_{k,n}} \frac{\delta net^j_{k,n}}{\delta w_{m,k,n}} \\ &= \eta(t^j - s^j_{k,n}) s^j_{k,n}(1 - s^j_{k,n}) s^j_{i,n} = \eta \delta_{k,n} s^j_{k-1,n}\end{aligned}$$

Where $s^j_{k-1,n}$ is the output from neuron n of layer $k-1$.

For neurons in hidden layers, we need to backpropagate the error through the layers in front:

$$\begin{aligned}\Delta w^j_{m,i,n} &= -\eta \left(\sum_p \frac{\delta E^j_{k,p}}{\delta s^j_{k,p}} \frac{\delta s^j_{k,p}}{\delta net^j_{k,p}} \frac{\delta net^j_{k,p}}{\delta s^j_{i,n}} \right) \frac{\delta s^j_{i,n}}{\delta net^j_{i,n}} \frac{\delta net^j_{i,n}}{\delta w_{m,i,n}} \\ &= \eta(\sum_p \delta_{kp} w_{m,k,p}) s^j_{in}(1 - s^j_{i,n}) x^j_i = \eta \delta_{i,n} x^j_i\end{aligned}$$

The intuition for this is that the neuron in the hidden layer will contribute its output to several neurons in the layer ahead. Thus, we need to sum the errors from the neurons of the front layer, propagated through the respective coefficients of those front neurons.

This is the *backpropagation algorithm*:

- Present the example to the MLP and activate all neurons, propagating the activation forward through the network.
- Compute the $\delta_{n,k}$ for each neuron n of layer k, starting from the output layer and then backpropagating the error through to the first layer.
- With the $\delta_{n,k}$ values .

With this algorithm and the MLP architecture shown on the left panel of Figure 8.7, we can train the network to classify the XOR function output. During training the two neurons on the hidden layer learn to transform the training set so that their outputs result in a linearly separable set that the neuron on the output layer can then separate.

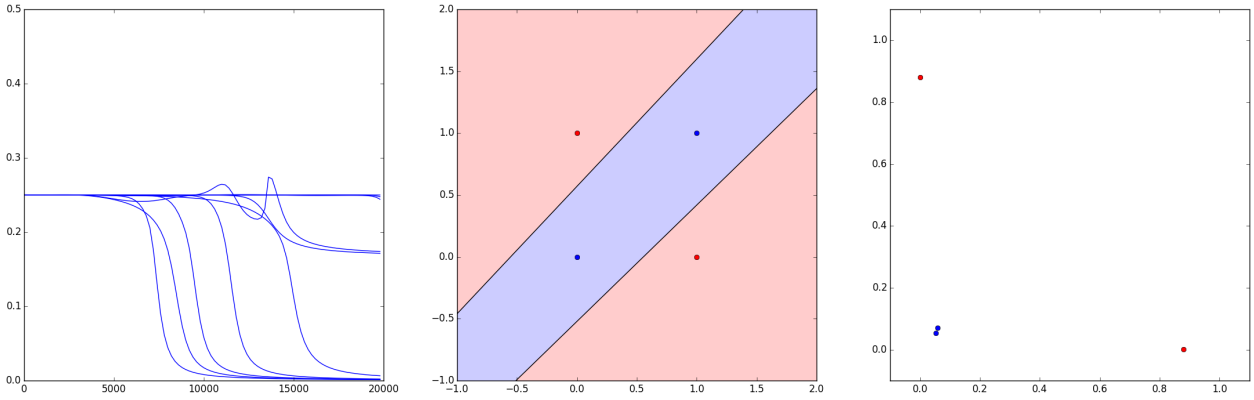

Figure 8.8: Training the MLP with one hidden layer for classifying the XOR function output. The first panel shows the training error over 10 training runs. Note that, due to the stochastic initialization and ordering of the examples presented, there are differences between different runs. The second panel shows the resulting classifier, successfully separating the classes. The third panel shows the output of the two neurons in the hidden layer of the network. This layer transforms the features of the training set making it linearly separable.

This ability to recode the features can be used explicitly in autoassociator networks. These networks are trained so that the output equals the input, while a hidden layer with a smaller number of neurons re-encodes the data. Figure 8.9 shows an example, from Mitchell [18], showing a MLP with 8 inputs, 8 output neurons and 3 neurons on the hidden layer. By forcing the output neuron activated to correspond to the input neuron set to 1, the hidden layer learns to recode the 8 possible values in combinations of three 0,1 values.

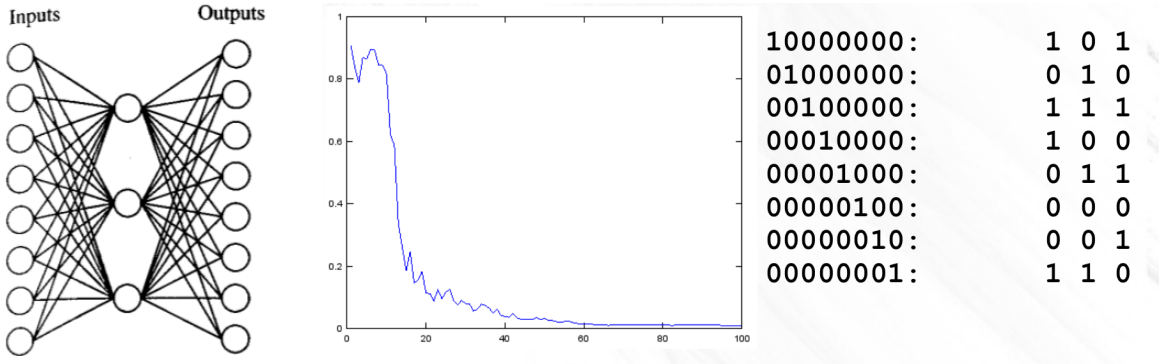

Figure 8.9: Autoassociator example. The network, shown on the left, was trained with the 8 different values consisting of one input set to 1 and the remainder set to 0, and forced to generate the same output. The hidden neurons recode the input into different combinations of neuron activations.

8.4 Training the Multilayer Perceptron

To train the MLP it is important to start with small, random weights, close to 0. This is because the sigmoidal activation functions saturate away from zero. It is also important to run the training process several times, since the training is not always exactly the same. Normalizing or standardizing the inputs

is also important, since input features at different scales will force the network to adjust weights at different rates.

To train the network, we present all training examples in a random order. One pass through all the training examples is one *epoch*. Then we repeat this process until the error converges or we detect overfitting. We can detect overfitting using cross-validation. This is also a form of regularization in training neural networks, as it allows us to stop training before the training error converges to a fixed value and thus avoid overfitting. Figure 8.10 illustrates this method.

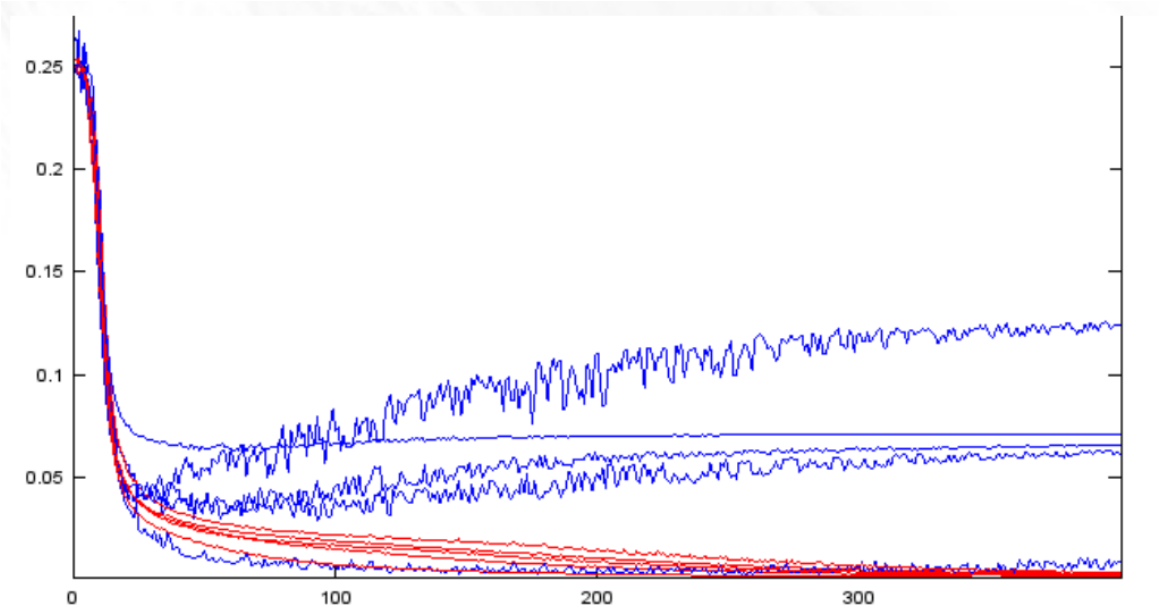

Figure 8.10: Validation (blue) and training error for five-fold cross-validation. Although training error keeps decreasing, it is best to stop training at epoch 40 to prevent overfitting.

Another form of regularization is to decay the coefficient weights by a small amount at each iteration, changing the update function to :

$$\Delta w_j = -\eta \frac{\delta E}{\delta w_j} - \lambda w_j$$

8.5 Further Reading

1. Alpaydin [2], Chapter 11
2. Mitchell [18], Chapter 4
3. Marsland [17], Chapter 3
4. (Bishop [4], Chapter 5)

Chapter 9

Support Vector Machines, part 1

Maximum margin classifier. Signed distance to decision frontier. Support Vectors and Support Vector Machine for linear classification

9.1 Maximum margin

In logistic regression and the perceptron we saw examples of separating different classes using a hyperplane. We saw that the squared error between the class and the distance to the hyperplane was not a good measure because it pulls the frontier towards the more distant points. By using a function that "squashes" the outputs away from the frontier, such as the logistic function, allowed us to find a better way of separating the classes. Figure 10.1 illustrates this difference. The left panel shows a linear discriminant computed by minimizing the squared errors loss function $e_{sq.}$, and the right panel the discriminant obtained with logistic regression, minimizing the loss function $E_{log.}$:

$$E_{sq.}(\widetilde{w}) = \sum_{j=1}^{N} (g(\vec{x_j}) - t_j)^2 \qquad E_{log.}(\widetilde{w}) = -\sum_{n=1}^{N} [t_n \ln g_n + (1 - t_n) \ln(1 - g_n)]$$

$$g_n = \frac{1}{1 + e^{-(\vec{w}^T \vec{x_n} + w_0)}}$$

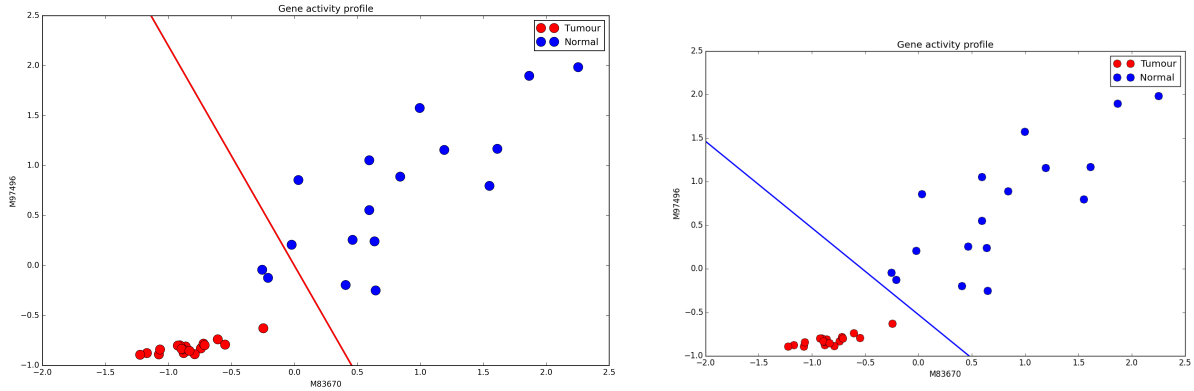

Figure 9.1: Gene activities for cancerous and normal cells. The linear discriminants were computed by least squared error (left panel) and logistic regression (right panel).

However, there is one disadvantage to an error function like the one used in logistic regression. While the quadratic error function has a well-defined minimum, because increasing the norm of vector $\widetilde{w}$ can make the decision function arbitrarily steep in the frontier, the frontier can be placed in any of a range of possible places. Figure 9.2 illustrates this problem. The left panel shows a series of results from the logistic regression minimization. Since the logistic function is flat away from the frontier, displacing the frontier makes little difference. This may result in the frontier being placed closer to some points, as shown in the right panel, increasing overfitting.

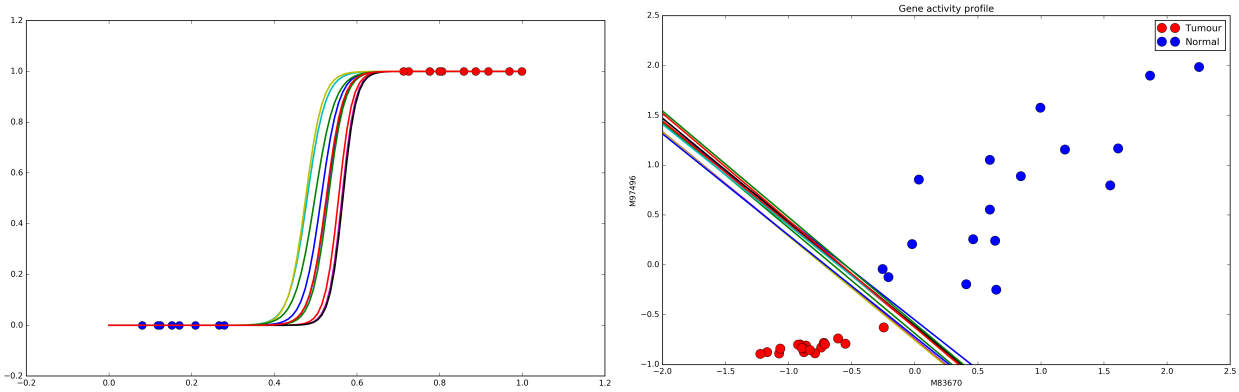

Figure 9.2: Logistic regression frontiers. Different runs of the same optimization result in different positions for the frontier because, if the logistic function is very steep at the frontier, placing it at different positions makes no difference for the loss function.

Regularization helps reduce this effect by forcing vector $\widetilde{w}$ to be shorter, smoothing the logistic function and forcing the frontier away from the closest points. Figure 9.3 shows the same logistic regression results as Figure 9.2, but this time using L2[1] regularization. This forces the optimization to always give the same result and always the same frontier, fixed farther away from the points.

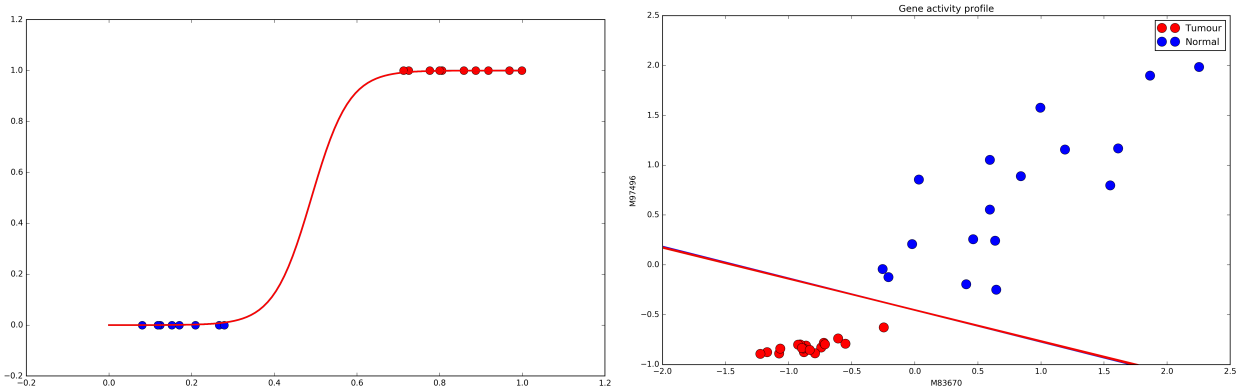

Figure 9.3: Logistic regression with L2 regularization, forcing the norm of $\widetilde{w}$ to be small, smooths the function and fixes the frontier away from the closest points.

This example shows the important concept of a *maximum margin classifier*. A *margin classifier* is a classifier that provides a measure of the distance between the frontier and the points closest to it. This is the *margin* of the classifier. A *maximum margin classifier* is a classifier that maximizes this distance. With logistic regression, we can approximate this using regularization, but this requires modifying the loss function to include the regularization term. The code below shows the loss function

[1] L2 regularization penalizes the square of the norm of $\widetilde{w}$

for the regularized logistic regression. Even though the regularization constant is small (0.00001), regularization always distorts the objective function of minimizing the error.

```
def log_cost(theta,X,y):
    coefs = np.zeros((len(theta),1))
    coefs[:,0] = theta
    sig_vals = logistic(np.dot(X,coefs))
    log_1 = np.log(sig_vals)*y
    log_0 = np.log(1-sig_vals)*(1-y)
    return -np.mean(log_0+log_1)+np.sum(coefs**2)*0.00001
```

A better option is to make margin maximization an explicit goal for our loss function. Figure 9.4 shows the margin, which is the distance between the frontier and the examples closest to it. These vectors are called the *support vectors*. To explicitly maximize the margin, we can consider the signed distance between a vector and the decision hyperplane:

$$r = \frac{\vec{w}^T x + w_0}{||\vec{w}||}$$

The value of r is positive on one side of the decision hyperplane and negative on the other, because of the value of the inner product $r = \vec{w}^T x + w_0$. Furthermore, r is invariant with respect to the norm of the vector defining the hyperplane, due to the division by $||\vec{w}||$.

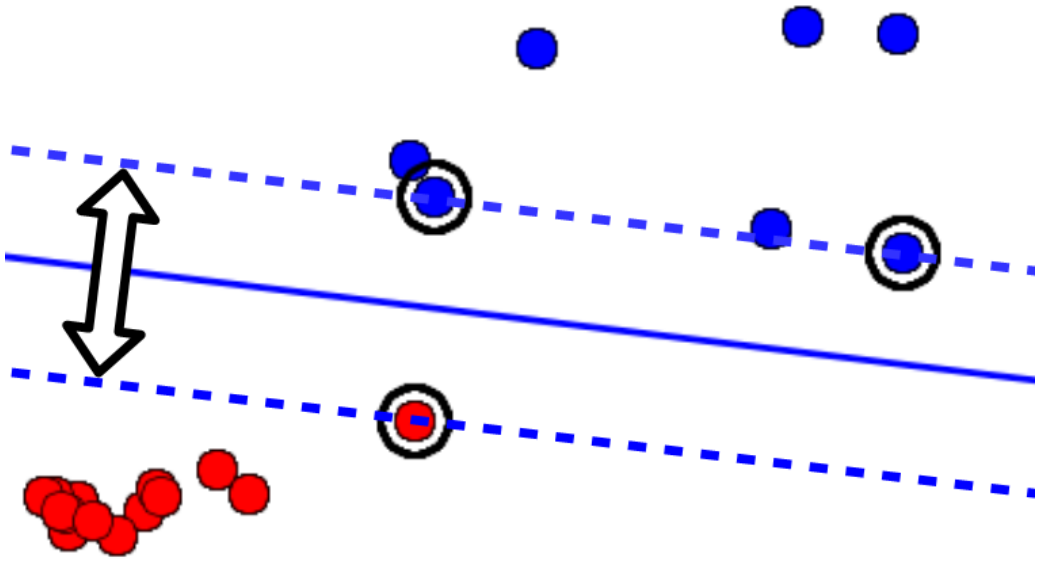

Figure 9.4: The margin is the distance to the points nearest to the decision frontier.

Now we can try to find the hyperplane that maximizes the minimum distance to points being classified:

$$\underset{\vec{w},b}{\text{argmax}} \left(\min_j \frac{y_j(\vec{w}^T x_j + w_0)}{||\vec{w}||} \right)$$

writing this loss function:

```
def closest_dist(ws, Xs, Ys):
    coefs = np.zeros((len(ws)-1,1))
    coefs[:,0] = ws.flatten()[:-1]
```

```
    dists = np.dot(Xs,coefs) + ws[2]
    norm = np.sqrt(ws[0]**2+ws[1]**2)
    return -np.min(dists * Ys / norm)

# load data
x0 = np.random.rand(3)
sol =  minimize(closest_dist, x0, args = (Xs,Ys))
```

Note the negative sign on the returned value because we are using the `minimize` function to find the maximum value. Despite the conceptual simplicity of this solution and the ease with which it can be implemented, unfortunately this does not work. As Figure 9.5 shows, the landscape of this loss function has discontinuous derivatives because, as the position of the discriminant hyperplane changes, it also changes which vectors are closest to this plane. Furthermore, the maximum value of the margin coincides with orientations for which several vectors are equidistant to the decision hyperplane, making the derivative discontinuous at the desired solution. Because of this, the optimization algorithm is unable to find the correct solution, as shown on the right panel.

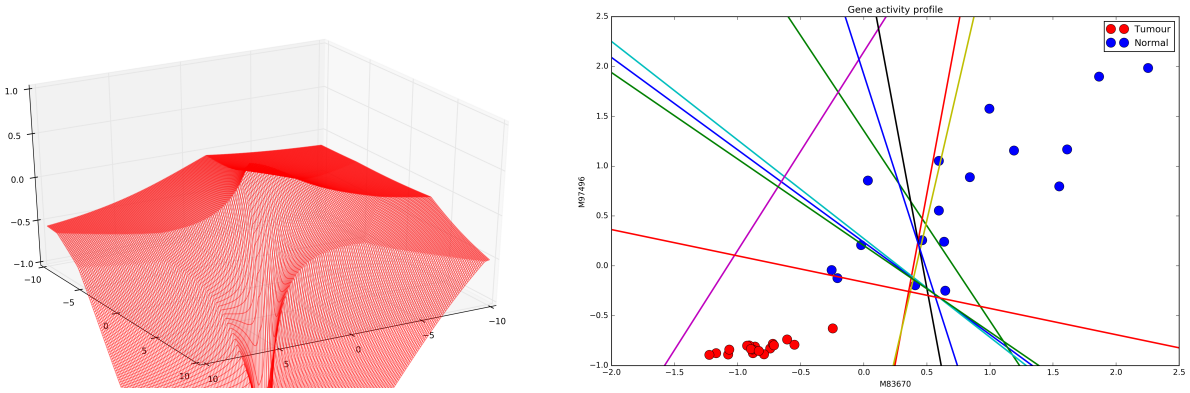

Figure 9.5: Landscape of the loss function for maximizing the minimum distance to the decision hyperplane (left panel) and the resulting hyperplanes due to the inability of the minimization algorithm to converge to the maximum margin.

To solve this problem we need a different approach.

9.2 Support Vector Machine

We start by noting that the normalized distance is invariant to scaling:

$$\frac{y_n(w^T x_n + w_0)}{||w||} = \frac{y_n(\beta \vec{w}^T x_n + \beta w_0)}{\beta ||\vec{w}||}$$

So we can impose this condition by making $\vec{w}$ and w_0 as large as necessary:

$$y_n(\vec{w}^T x_n + w_0) \geq 1, \forall n \in N$$

Subject to this condition, the problem of maximizing the margin is equivalent to the problem of minimizing the norm $||\vec{w}||$. As long as the condition above is not violated, shrinking the size of $\vec{w}$ means we are effectively increasing the distance between the discriminant and the closest points. More conveniently, we can minimize a quadratic function of the norm $||\vec{w}$, since minimizing quadratic functions is computationally more convenient.

$$\underset{\vec{w},b}{\text{argmax}} \left(\min_j \frac{y_j(\vec{w}^T x_j + w_0)}{||\vec{w}||} \right) = \arg\min_{\vec{w},w_0} \frac{1}{2}||\vec{w}||^2$$

Note that w_0 is determined by the constraint. This is thus a constraint optimization problem. One method for solving constraint optimization problems is to use *Lagrange multipliers*. To illustrate the approach, consider the following example:

$$\arg\max_{x,y} (1 - x^2 - y^2) \qquad s.t. x - y - 1 = 0$$

At the maximum along the line defining the constraint, the component of the function's derivative that is parallel to the constraint line must be zero, because otherwise this would not be a local maximum. This means that, at this point, the constraint line is tangent to a contour line of the objective function. Figure 9.6 illustrates this.

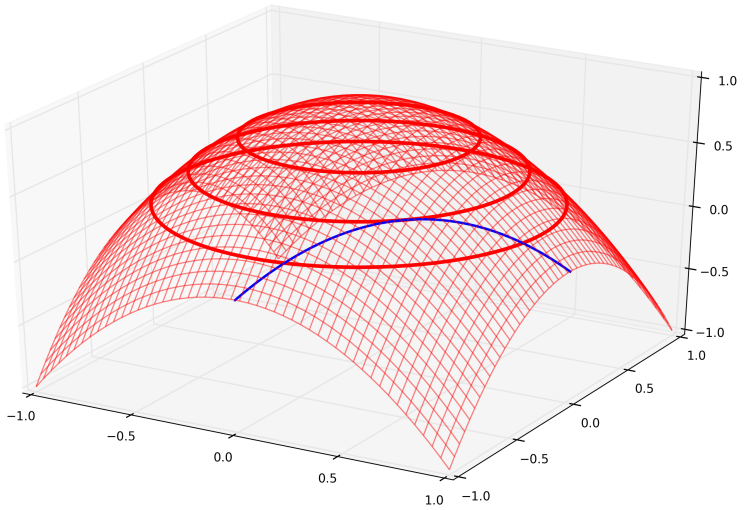

Figure 9.6: Objective function surface (in red) and the constraint line (blue). The red lines depict the contour lines of the objective function.

Since $g(x, y) = 0$ is a contour line of g, if $f(x, y)$ is a maximum subject to $g(x, y) = 0$, then the contour line of $f(x, y)$ is parallel to the contour line of $g(x, y)$. And if the contour lines are parallel then the gradient vectors are also parallel, because the gradient must be perpendicular to the contour line. So we can write:

$$\vec{\nabla}_{x,y} f(x, y) = -\alpha \vec{\nabla}_{x,y} g(x, y)$$

The negative sign is conventional and α is a *Lagrange multiplier*. We combine these in the *Lagrangian function*:

$$\mathcal{L}(x, y, \alpha) = f(x, y) + \alpha g(x, y)$$

and solve:

$$\vec{\nabla}_{x,y,\alpha} \mathcal{L}(x, y, \alpha) = 0$$

to find the critical points of the *Lagrangian*, among which the constrained optimum can be found. In our example:

$$\vec{\nabla}_{x,y,\alpha}\left(1 - x^2 - y^2 + \alpha(x - y - 1)\right) = 0$$

Can be solved by

$$\frac{\delta\mathcal{L}}{\delta x} = -2x + \alpha \qquad \frac{\delta\mathcal{L}}{\delta y} = -2y - \alpha \qquad \frac{\delta\mathcal{L}}{\delta\alpha} = x - y \breve{} 1$$

$$x - y - 1 = 0 \Leftrightarrow x = y + 1 \qquad \alpha = 2x\,, \alpha = -2y \Leftrightarrow x = -y$$

The solution is thus $\{0.5, -0.5\}$. Applying the same method to the problem of maximizing the margin of the classifier:

$$\arg\min_{w,w_0} \frac{1}{2}||\vec{w}||^2 \qquad s.t. y_n(\vec{w}^T\vec{x}_n + w_0) \geq 1, \forall n \in N$$

Noting that

$$y_n(\vec{w}^T\vec{x}_n + w_0) \geq 1 \iff -\left((y_n(\vec{w}^T x_n + w_0) - 1\right) \leq 0$$

we obtain the following *Lagrangian* :

$$\mathcal{L}(\vec{w}, w_0, \vec{\alpha}) = \frac{1}{2}||\vec{w}||^2 - \sum_{n=1}^{N} \alpha_n \left(y_n(\vec{w}^T x_n + w_0) - 1\right)$$

We want to minimize the function with respect to vector $\vec{w}$ and b, while obtaining the maximum with respect to the α_n multipliers. Since at the critical point the derivatives with respect to $\vec{w}$ and w_0 are 0, we can write:

$$\frac{\delta\mathcal{L}}{\delta\vec{w}} = 0 \Leftrightarrow \vec{w} = \sum_{n=1}^{N} \alpha_n y_n \vec{x}_n \qquad \frac{\delta\mathcal{L}}{\delta w_0} = 0 \Leftrightarrow \sum_{n=1}^{N} \alpha_n y_n = 0$$

Replacing, we obtain the *dual representation* of our original problem. In optimization problems, duality is a relation between two different perspectives on the problem, solving for different sets of variables. In general, the *dual* of an optimization problem only provides a bound on the optimal value but, in this case, solving the dual solves the original problem. So now we solve this *dual representation* of our problem as a function of the *lagrangian multipliers*:

$$\tilde{\mathcal{L}}(\vec{\alpha}) = \sum_{n=1}^{N} \alpha_n - \frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{N} \alpha_n\alpha_m y_n y_m \vec{x}_n^T\vec{x}_m$$

$$\sum_{n=1}^{N} \alpha_n y_n = 0 \qquad \alpha_n \geq 0$$

This can be solved by standard quadratic programming algorithms.

9.3 Implementing a Support Vector Machine

As an example, we'll see how to implement a SVM using the `optimize` function from the `scipy` library. First, we compute the matrix with the inner products of all pairs of training vectors multiplied by their respective classes:

$$H = \sum_{n=1}^{N}\sum_{m=1}^{N} y_n y_m \vec{x}_n^T \vec{x}_m$$

```
def H_matrix(X,Y):
    H = np.zeros((X.shape[0],X.shape[0]))
    for row in range(X.shape[0]):
        for col in range(X.shape[0]):
            H[row,col] = np.dot(X[row,:],X[col,:])*Y[row]*Y[col]
    return H
```

Now we define the function to maximize:

$$\arg\max_{\vec{\alpha}} \sum_{n=1}^{N} \alpha_n - \frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{N} \alpha_n \alpha_m y_n y_m \vec{x}_n^T \vec{x}_m$$

Intuitively, we can see that α_n will be zero (all *lagrangian multipliers* are non-negative numbers) for all vectors that are surrounded by vectors of the same class, because for these the inner products with vectors of the same class will have a greater weight in the sum, and these contribute positively to the total because the products of the class labels $y_n y_m$ will be positive. Conversely, for vectors close to vectors of the opposite class, there will be a non-zero optimal value for α_n that balances the increase of the sum of the α values and the penalty given to the sum of the inner products. However, if a point has too many neighbours of the other class, the inner products sum will be negative and the α_n value will tend towards infinity, so no solution can be found. This happens if the classes are not linearly separable.

Since we are using the `minimize` function, we need to change the sign of the result. It is also useful to provide the optimization algorithm with the jacobian matrix, which consists of the derivatives of our function with respect to each α_n. This improves the convergence of the algorithm.

```
def loss(alphas):
    return 0.5 * np.dot(alphas.T, np.dot(H, alphas)) - np.sum(alphas)

def jac(alphas):
    return np.dot(alphas.T,H)-np.ones(alphas.shape[0])
```

Now we set up the constraints and minimize the target function using the Sequential Least Squares Programming method (SLSQP).

```
H = H_matrix(Xs,Ys)
A = Ys[:,0] # sum of alphas is zero
cons = {'type':'eq',
        'fun':lambda alphas: np.dot(A,alphas),
        'jac':lambda alphas: A}
bounds = [(0,None)]*Xs.shape[0] #alpha>=0
x0 = np.random.rand(Xs.shape[0])
sol =  minimize(loss, x0, jac=jac, constraints=cons, method='SLSQP', bounds = bounds)
```

For the `minimize` function, the constraints are specified in the dictionary with the function and derivatives for the constraint line setting $\sum_{n=1}^{N} \alpha_n y_n = 0$, which corresponds to the inner product of thevector of α values and the classes of the respective points. The constraint $\alpha_n \geq 0$ is specified in the `bounds` variable.

For all examples that are distant from the margins, the α values are zero. The *support vectors*, those examples that lie at the margins, have an α value greater than zero and can be easily identified:

```
svs = sol.x>0.001
print svs
[False False False False False False  True False False False False False
 False False False False False False False False False False False  True
 False False False False False False False False False False False  True]
```

Now we can compute $\vec{w}$ and b from the *support vectors* (for b, we can average over all support vectors).

$$\vec{w} = \sum_{n=1}^{N} \alpha_n y_n \vec{x}_n \qquad b = y_n - \vec{w}^T \vec{x}_n$$

```
def svm_coefs(X,Y,alphas):
    w = np.sum(alphas*Y*X.T,axis = 1)[:,np.newaxis]
    b = np.mean(Y-np.dot(X,w))
    coefs = np.zeros(len(w)+1)
    coefs[-1] = b
    coefs[:-1] = w.flatten()
    return coefs

coefs = svm_coefs(Xs[svs,:],Ys[svs,0],sol.x[svs])
```

Figure 9.7 shows the resulting hyperplane and the support vectors found.

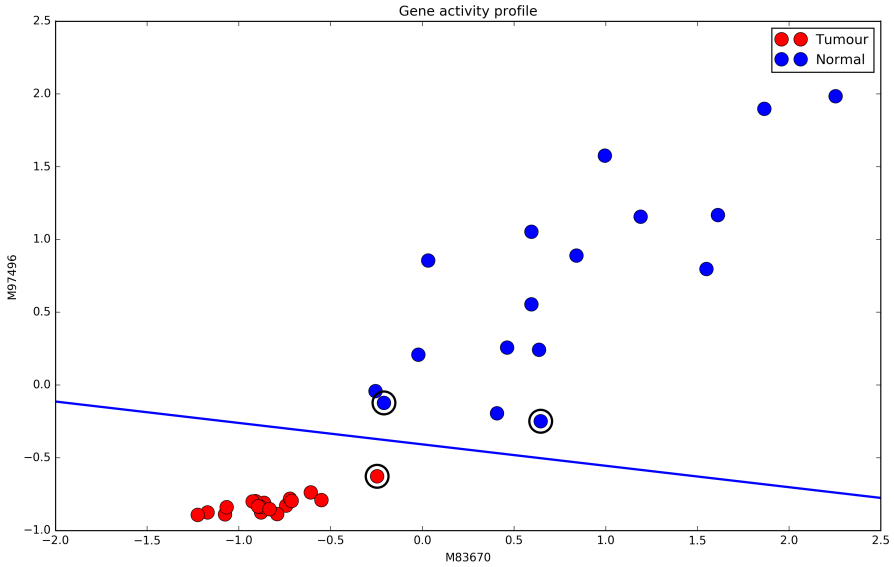

Figure 9.7: SVM classifier. The decision hyperplane is represented in blue and the support vectors are outlined with a black circle.

9.4 Further Reading

1. Alpaydin [2], Sections 13.1 and 13.2
2. Marsland [17], Section 5.1

Chapter 10

Support Vector Machines, part 2

Support Vector Machine: soft margins and the kernel trick. Regularization of SVM

10.1 Soft Margins

In the previous chapter, we derived this *dual problem* from the problem of minimizing the norm of the hyperplane vector subject to the constraint that the margin would be at least 1:

$$\arg\max_{\vec{\alpha}} \sum_{n=1}^{N} \alpha_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} \alpha_n \alpha_m y_n y_m \vec{x}_n^T \vec{x}_m$$

$$\sum_{n=1}^{N} \alpha_n y_n = 0 \qquad \alpha_n \geq 0$$

However, this is only possible if the data sets are linearly separable. Otherwise, the constraints are incompatible and the problem has no solution, as illustrated on Figure 10.1. An intuitive way of understanding this is to note that, if vectors $\vec{x}$ are surrounded by neighbours of another class, then the corresponding α values can rise to infinity to maximize the target function. This means the function no longer has a maximum value.

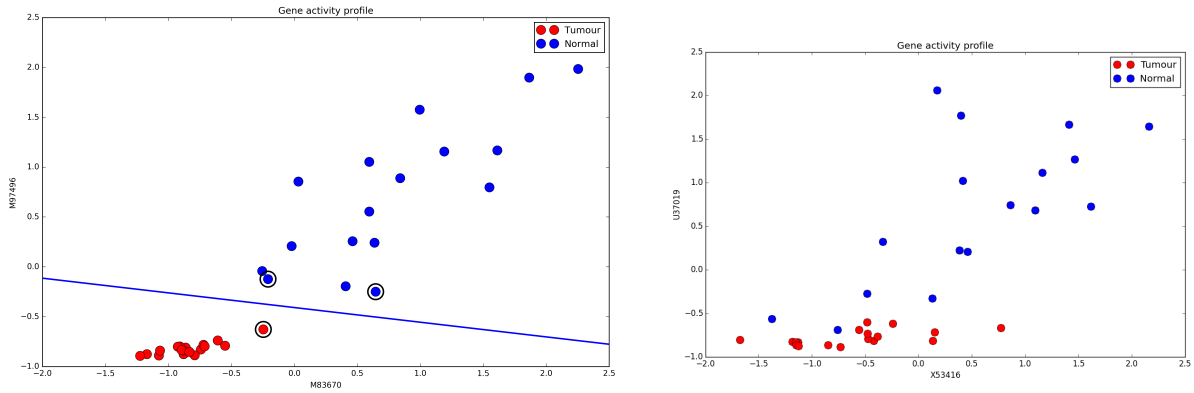

Figure 10.1: The left panel shows the decision frontier of a linear SVM. The right panel shows an example of data that is not linearly separable.

To solve this problem, we can add a slack variable ξ_n for each vector, a positive value representing the distance between the vector and the inside of the margin, or zero if the vector is not inside the margin:

$$y_n(\vec{w}^T x_n + b) \geq 1 - \xi_n, \forall n \in N \qquad \xi \geq 0$$

If $1 > \xi_n > 0$, then the vector $\vec{x}$ is inside the margin; if $\xi_n > 1$, then the vector $\vec{x}$ is on the wrong side of the decision hyperplane. This allows vectors to penetrate the margins. However, we want to minimize the violation of the margin constraint, so now we want to minimize $||\vec{w}||^2$ but penalizing violations to the margin:

$$\arg\min C \sum_{n=1}^{N} \xi_n + \frac{1}{2}||\vec{w}||^2$$

For the new lagrangian, we need additional lagrangian multipliers for the ξ variables, given the constraint that they must be at zero or greater:

$$\mathcal{L}(\vec{w}, b, \vec{\alpha}, \vec{\mu}, \vec{\xi}) = \frac{1}{2}||\vec{w}||^2 + C\sum_{n=1}^{N} \xi_n - C\sum_{n=1}^{N} \alpha_n \left(y_n(\vec{w}^T\vec{x} + b) - 1 + \xi_n\right) - \sum_{n=1}^{N} \mu_n \xi_n$$

Setting the derivatives to 0 we obtain the same dual problem, but the derivative of the lagrangian as a function of the ξ slack variables forces the α parameters to be less than C:

$$\arg\max_{\vec{\alpha}} \sum_{n=1}^{N} \alpha_n - \frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{N} \alpha_n\alpha_m y_n y_m \vec{x}_n^T \vec{x}_m$$

$$\sum_{n=1}^{N} \alpha_n y_n = 0$$

$$\frac{\delta\mathcal{L}}{\delta\xi_n} = 0 \Leftrightarrow C - \alpha_n - \mu_n = 0 \Leftrightarrow 0 \leq \alpha_n \leq C, \qquad \sum_{n=1}^{N} \alpha_n y_n = 0$$

So, to fit the SVM with soft margins, we just need to add the C parameter to the upper bounds of the α values:

```
H = H_matrix(Xs,Ys)
A = Ys[:,0] # sum of alphas is zero
cons = {'type':'eq',
        'fun':lambda alphas: np.dot(A,alphas),
        'jac':lambda alphas: A}
bounds = [(0,C)]*Xs.shape[0] #alpha>=0
x0 = np.random.rand(Xs.shape[0])
sol =  minimize(loss, x0, jac=jac, constraints=cons, method='SLSQP', bounds = bounds)
```

With this change, it is now possible to compute a SVM classifier for a data set in which the classes have a slight overlap. Figure 10.2 shows the result. The thin green lines represent the margins. All support vectors – the vectors for which the α values are greater than zero – are marked with a circle. Those with red circles are inside the margins, corresponding to ξ values greater than zero and α values maximized to C

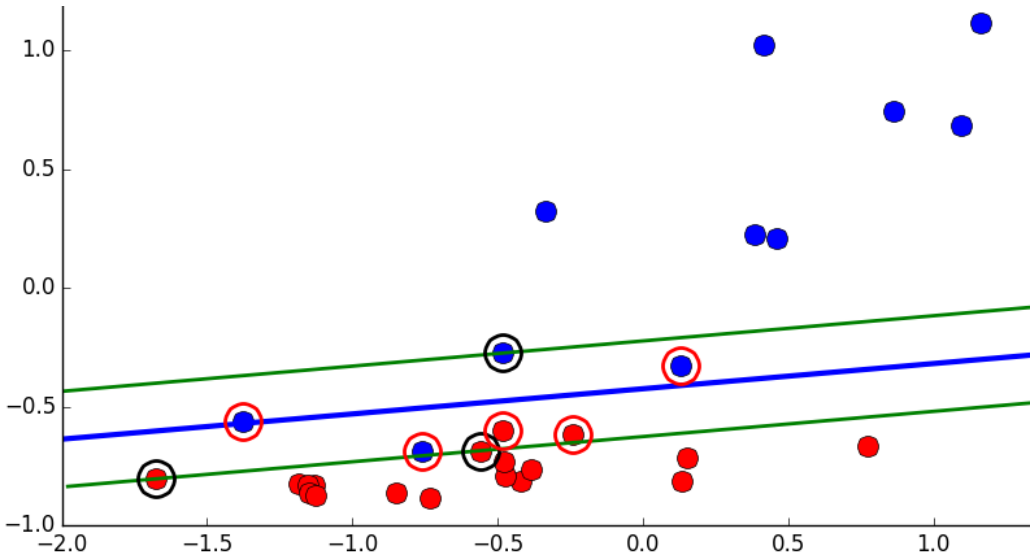

Figure 10.2: Soft-margin separation with SVM. Vectors indicated with circles are support vectors. Those inside red circles are support vectors inside the margins, for which $\alpha = C$.

Note that, in this case, to compute the w_0 parameter we can only use the support vectors that do not penetrate the margins (*i.e.* those for which $alpha < C$). For those that lie inside the margins ($\alpha = C$) the equation $y_n(\vec{w}^T\vec{x}_n + w_0) = 1$ is not valid. So, for computing w_0 we average $y_n - \vec{w}^T\vec{x}_n$ using only those support vectors for which $0 < \alpha_n < 0$.

10.2 Non-linear separation and the Kernel Trick

Soft margins can solve slight overlaps, but sets that are not linearly separable we'll generally need a different approach.

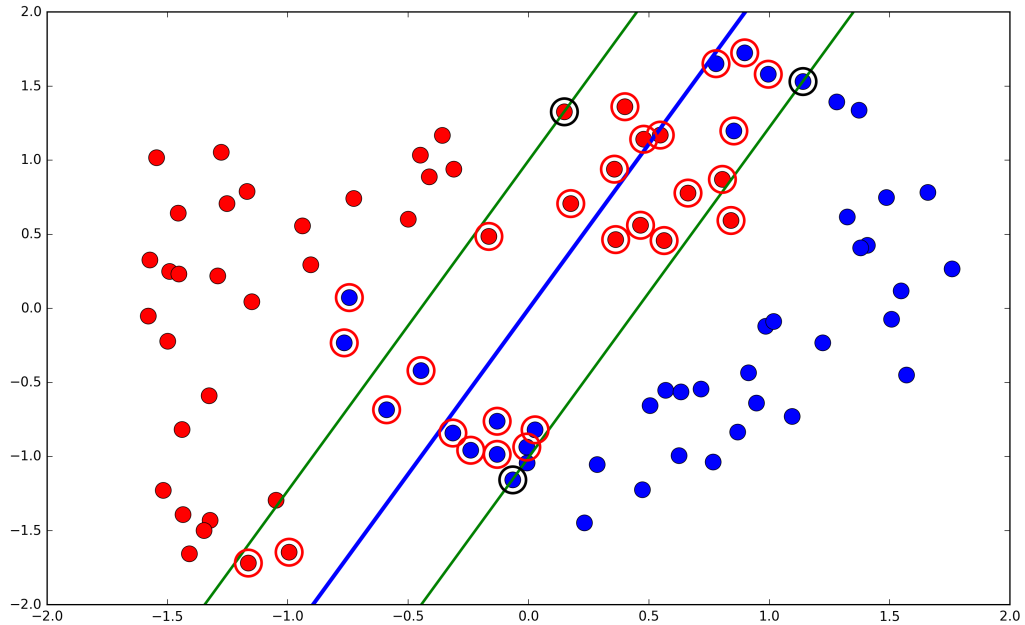

Figure 10.3: For data that is not linearly separable, soft-margins alone are generally not a good solution.

As we saw several times before, the way to classify sets that are not linearly separable with linear classifiers is to use a representation of the data in higher dimensions. With SVM, this is easy to do

with *kernel functions*. First, we note that all the inner products of the pairs of training vectors can be precomputed, as we did in the last chapter:

$$\arg\max_{\vec{\alpha}} \sum_{n=1}^{N} \alpha_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} \alpha_n \alpha_m y_n y_m \boxed{\vec{x}_n^T \vec{x}_m}$$

If we expand the training vectors to higher dimensional representations, we could compute the inner products of these expanded vectors. However, there is an even better way to do this. Since all we need are the inner products and not the vectors themselves, we can use *kernel functions*. A *kernel function* is a function that gives us the inner product of the transformed vectors as a function of the original vectors:

$$K(\vec{x}_1, \vec{x}_2) = \phi(\vec{x}_1)^T \phi(\vec{x}_2)$$

For example, this function ϕ transforms a two-dimensional vector into a six-dimensional vector:

$$\phi(\vec{x}) = [1, \sqrt{2}x_1, \sqrt{2}x_2, \sqrt{2}x_1 x_2, x_1^2, x_2^2]^T$$

But, in this case, $\phi(\vec{x_1})^T \phi(\vec{x_1}) = (\vec{x_1}^T \vec{x_2} + 1)^2$, and so we have a *kernel function* in this case, which is $(\vec{x_1}^T \vec{x_1} + 1)^2$. More generally, for degree n polynomial expansions of this sort, the *kernel function* is:

$$K_{\phi^n}(\vec{x_1}, \vec{x_2}) = (\vec{x_1}^T \vec{x_2} + c)^n$$

So we can solve the original problem, but using a *kernel function* that implicitly expands the data to a higher-dimensional space:

$$\arg\max_{\vec{\alpha}} \sum_{n=1}^{N} \alpha_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} \alpha_n \alpha_m y_n y_m K(\vec{x_n}, \vec{x_m})$$

To classify a new vector $\vec{x_t}$, once the SVM is trained, we cannot compute $\vec{w}$ because$\vec{w}$ will define a hyperplane on the higher-dimensional space where the *kernel* implicitly projects the original data. So, instead, we compute the class of $\vec{x_t}$ using the support vectors:

$$\vec{w}^T \phi(\vec{x_t}) + w_0 = \sum_{n=1}^{N} \alpha_n y_n K(\vec{x_n}, \vec{x_t})$$

For example, we can use a polynomial kernel of degree d, $K_{\phi^d}(\vec{x_1}, \vec{x_2}) = (\vec{x_1}^T \vec{x_2} + 1)^d$:

```
def H_poly(X,Y,n):
    H = np.zeros((X.shape[0],X.shape[0]))
    for row in range(X.shape[0]):
        for col in range(X.shape[0]):
            k = (np.dot(X[row,:],X[col,:])+1)**n
            H[row,col] = k*Y[row]*Y[col]
    return H
```

And to classify a new point we use the support vectors:

$$\vec{w}^T \phi(\vec{x_t}) + w_0 = \sum_{n=1}^{N} \alpha_n y_n K_{\phi^d}(\vec{x_1}, \vec{x_2})$$

```
def poly_k_class(X,alphas,Y,xt,d):
    s = 0
    for ix in range(len(alphas)):
        s = s + (np.dot(X[ix,:],xt)+1)**d*Y[ix]*alphas[ix]
    return s
```

However, in practice we will not implement this ourselves. This code is just to illustrate the computation. It is best to use an optimized implementation of a SVM, such as the one provided in the `sklearn` library. To use the SVM classifier from `sklearn`, we just need to specify the kernel type and corresponding parameters to the constructor of the `SVC` class (support vector classifier):

```
from sklearn import svm
#load and standardize
sv = svm.SVC(C=C,kernel = 'poly', degree = poly, coef0 = 1)
sv.fit(Xs,Ys[:,0])
```

Figure 11.7 shows the result of fitting the SVM with a third degree polynomial kernel. Using a lower value of C places a lower upper limit on the penalty for margin violations. This leads to several support vectors being placed inside the margin (thus maximizing the α values to C). A higher value of C results in greater penalties for margin violations and, in this case, with $C = 1000$ no support vectors are placed inside the margins.

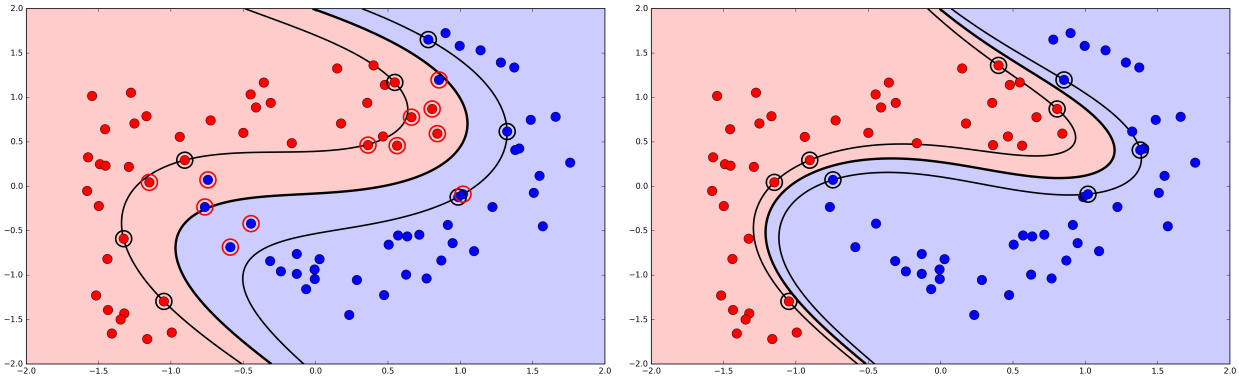

Figure 10.4: SVM trained with a third degree polynomial kernel. The SVM on the left panel was trained with $C = 1$, and the right with $C = 1000$.

So we can see C as a regularization parameter, with lower C values corresponding to higher regularization, simplifying the decision surface at the cost of allowing more errors. Figure 10.5 shows another example, this time using a Gaussian kernel, also known as Gaussian Radial Basis Function kernel, or RBF:

$$K(\vec{x_1}, \vec{x_2}) = e^{\frac{-||\vec{x_1}-\vec{x_2}||^2}{2\sigma^2}}$$

In Scikit Learn, $1/2\sigma^2$ is combined into the γ parameter in $K(\vec{x_1}, \vec{x_2}) = e^{-\gamma||\vec{x_1}-\vec{x_2}||^2}$.

```
from sklearn import svm
#load and standardize
sv = svm.SVC(C=C,kernel = 'rbf', gamma=gamma)
sv.fit(Xs,Ys[:,0])
```

Figure 10.5 shows different combinations of C and γ values. The top panels show the results of using $\gamma = 0.01$ and $\gamma = 2$ with $C = 1$. A higher γ value makes the RBF kernel weigh nearby points

more strongly, making the decision frontier conform more tightly to the two classes; a lower γ value broadens the radius of the RBF kernel smoothing the frontier. The bottom panels, with $C = 1000$ show the same effect but with less regularization.

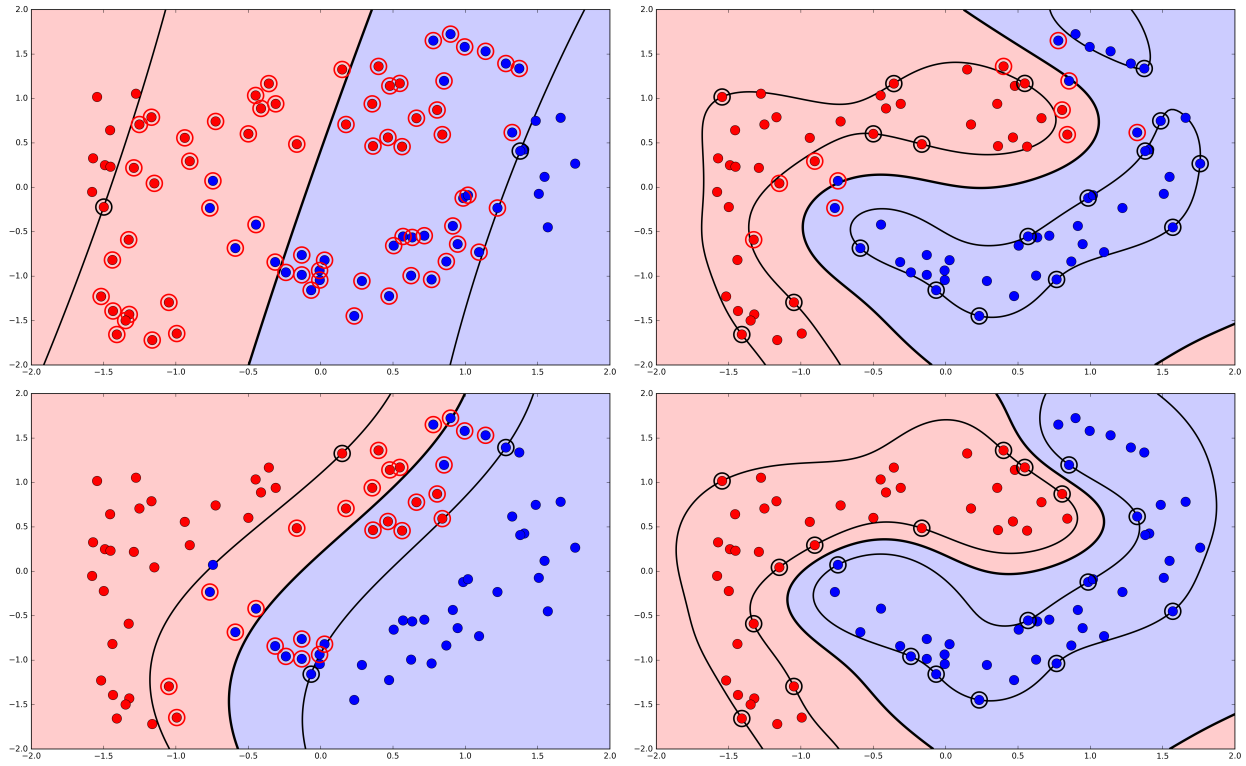

Figure 10.5: SVM trained with a RBF kernel. In the top panels, the SVM was trained with $C = 1$, and $C = 1000$ in the bottom panels. The panels on the left show SVM trained with $\gamma = 0.01$, those on the right with $\gamma = 2$.

10.3 Further Reading

1. Alpaydin [2], Sections 13.1 - 13.8
2. Marsland [17], Chapter 5
3. Bishop [4], Section 7.1

Chapter 11

Multiclass and Bias-Variance decomposition

Multiclass classification. Bootstrapping, Bias-Variance decomposition

11.1 Multiclass classification

So far we have always focused on binary classification but classification problems with more than two classes are common. A classical example is the classification of flowers of three species of the *Iris* genus: *Iris setosa*, *Iris versicolor* and *Iris virginica*, shown in Figure 11.1. The data set describes each flower with four features: sepal length and width and petal length and width[1].

Figure 11.1: Iris flowers: setosa, versicolor and virginica. Images CC BY-SA: Szczecinkowaty; Gordon abd Robertson; Mayfield

For classifiers like Naïve Bayes or k-Nearest Neighbours the number of classes makes no difference, since the classifier is used in exactly the same way. For Naïve Bayes, we choose the class that maximizes the conditional probability of the feature values:

$$C^{Nave\ Bayes} = \underset{k\in\{0,1,\ldots,K\}}{\operatorname{argmax}} \ln p(C_k) + \sum_{j=1}^{N} \ln p(x_j|C_k)$$

and for k-Nearest Neighbours we classify each new point according to the majority in its k-neighbourhood. Figure 11.2 illustrates the data set and its use for creating a k-NN classifier.

[1]The data set can be downloaded from the MIST repository: https://archive.ics.uci.edu/ml/datasets/Iris

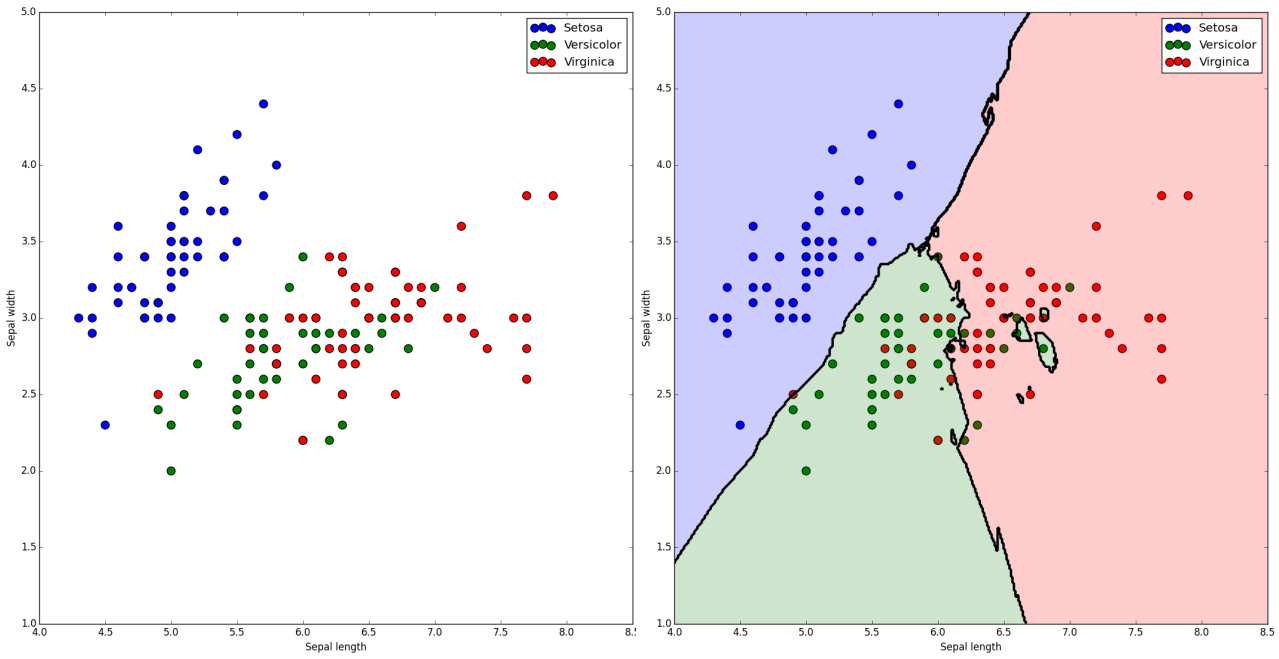

Figure 11.2: The left panel shows the Iris data set projected on the sepal length and width features. The right panel shows the classification with k-NN. Each point is classified according to the majority of the classes of neighbouring points.

However, for classifiers based on binary discriminant functions, like Logistic Regression, perceptrons or SVM, extension to more than two classes requires some meta-algorithm to obtain the necessary binary discriminants. One possible way of separating K classes is to train $K-1$ binary classifiers, each one to discriminate between one class and all other examples. This is an example of a *one versus the rest* classification scheme. A point is attributed to the class corresponding to the classifier that identifies it as being in the classifier's class, or to class K if none of the $K-1$ classifiers identifies it. Figure11.3 shows this process. One problem with the $K-1$ *one versus the rest* classification scheme is that there are ambiguous results wherever classifiers overlap. In this example, there are points that are classified both as *setosa* and *versicolor*.

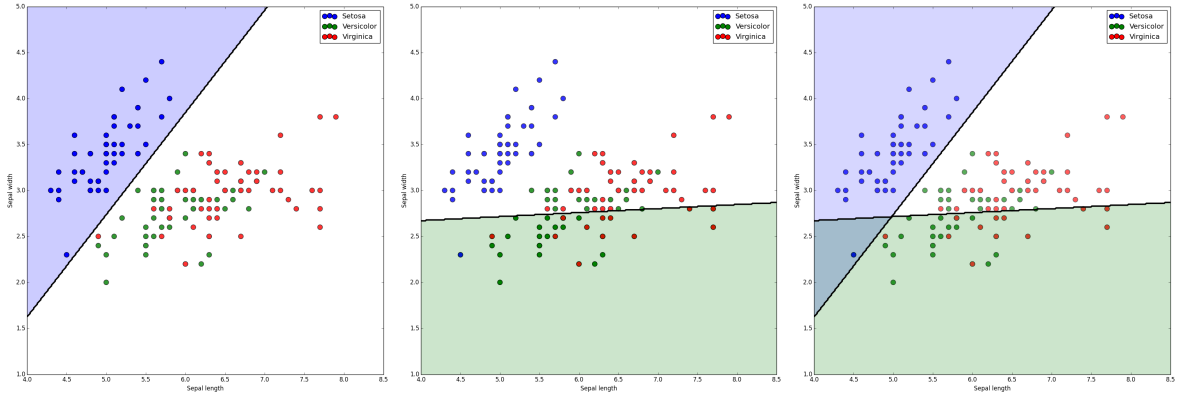

Figure 11.3: One versus the rest with K-1 classifiers. The first two classifiers distinguish, respectively, setosa and versicolor examples from all others. The last panel shows the final classification.

An alternative is to train binary classifiers to distinguish between all pairs of classes by training $K(K-1)/2$ classifiers and then classifying each new example with a majority vote, attributing it to the class with the largest number of votes among the classifiers. However, with this approach there are also ambiguous classifications whenever there is an equal number of votes for more than one class.

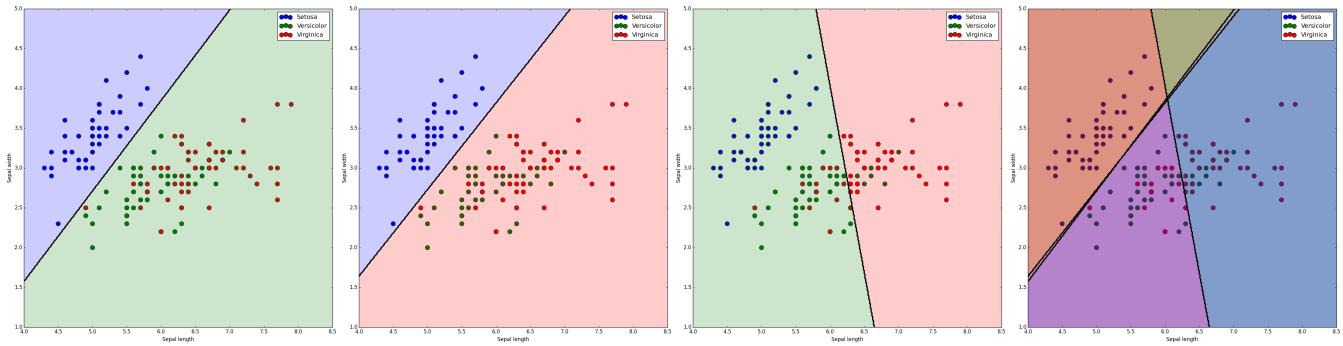

Figure 11.4: One versus the one classification schemes. After training $K(K-1)/2$ classifiers, points are classified by majority vote.

A better alternative is to use a *one versus the rest* classification scheme with K classifiers. If each classifier can provide a value for the decision function, points can be classified according to the maximum of the decision functions of the K classifiers. This solves the problem of ambiguous classification, as illustrated in Fig 11.5. However, for the *one versus the rest* scheme it is necessary to train each classifier with an unbalanced sample in which the majority of points fall outside the respective class. For example, if our training set has 10 evenly balanced classes, then each of the 10 classifiers will have only 10% of the points in the positive class and 90% in the negative class. Furthermore, the decision function values for the different one-vs-rest classifiers may not be directly comparable, and these differences may affect the performance of this multiclass classification heuristic.

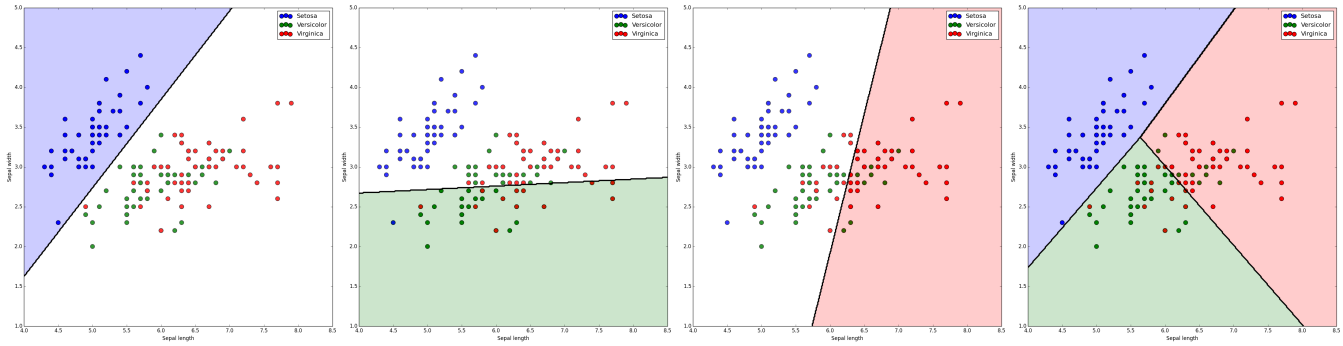

Figure 11.5: One versus the rest classification scheme with K classifiers. Points are classified by the maximum value of the decision function.

Some classifiers allow specific alternatives to multiclass classification. For example, Logistic Regression can be extended to multiclass classification by fitting K discriminant hyperplanes simultaneously, by using the *cross entropy* of all classes and predictions considering, for each of the K discriminants, that the points belong to class 1 if they are in class k and to class 0 otherwise:

$$p(T|w_1,...,w_K) = \prod_{n=1}^{N}\prod_{k=1}^{K} p(C_k|\phi_n)^{t_{nk}} = \prod_{n=1}^{N}\prod_{k=1}^{K} y_{nk}^{t_{nk}}$$

In this expression, the t_nk matrix gives this *one vs rest* classes, assigning a 1 to all elements in class k and 0 otherwise. In practice, we minimize the logarithm of the *cross entropy* as an error function.

$$E(w_1,...,w_K) = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_{nk} \ln y_{nk}$$

With the `sklearn` library, we can use either the *one vs rest* classification scheme (`'ovr'`) or the *cross entropy* one (`'multinomial'`) in the `LogisticRegression` class:

```
from sklearn.linear_model import LogisticRegression

#One versus rest, max
logreg = LogisticRegression(C=1e5,multi_class='ovr')
logreg.fit(X, Y)

#Cross entropy
logreg = LogisticRegression(C=1e5,multi_class='multinomial')
logreg.fit(X, Y)
```

The multilayer perceptron also can be easily adapted to multiclass classification by having one output neuron for each class and training the MLP to output a 1 on the neuron corresponding to the class of the example and a 0 on all other output neurons.

The `sklearn` library offers a useful class to perform *one versus rest* classification by training K classifiers and classifying each example according to the maximum of the decision function:

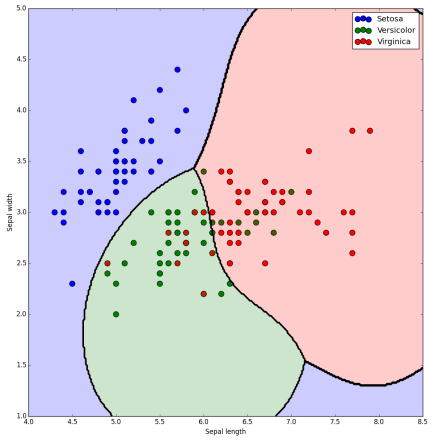

Figure 11.6: One-vs-rest classification of the Iris data set using SVM.

```
from sklearn.multiclass import OneVsRestClassifier

ovr = OneVsRestClassifier(SVC(kernel='rbf',
                              gamma=0.7, C=10))
ovr.fit(X, Y)
ovr.predict(test_set)
```

To use this class, we need only provide it with the class of the binary classifier, which must implement the `fit` and `decision_function` methods. The `fit` method of `OneVsRestClassifier` generates K classifiers, training each to distinguish one class from all others. Then the `predict` method returns the class corresponding to the classifier that outputs the largest value in the `decision_function`. Figure 11.6 shows the result of this process using SVM on the Iris dataset.

11.2 Bias and Variance

Statistically, *bias* is the difference between the expected value of an estimator and the true value being estimated. Thus, the *bias* of a model at some point is the difference between the true value and the expected prediction of the model for that point. The *bias* for the model is the average of the *bias* values measured for all points::

$$bias_n = (\bar{y}(x_n) - t_n)^2 \qquad bias = \frac{1}{N}\sum_{n=1}^{N}(\bar{y}(x_n) - t_n)^2$$

Figure 11.7 shows an example of a model that cannot adequately fit the data. The estimates for the point marked as a large blue circle are all tendentiously above the true value and thus there is a difference between the average and the true value.

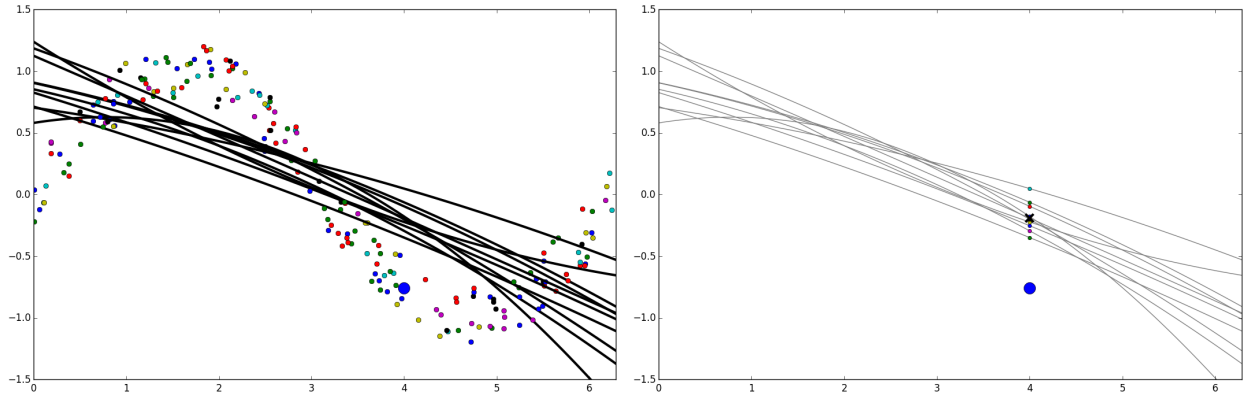

Figure 11.7: This model cannot adjust to the data and thus has a large *bias* in some points.

In statistics, variance is a measure of the dispersion of values. Applying this concept to a regression model, the *variance* of the model at some point is the expected variance of the predicted values for that point when the model is trained over any data set. The *variance* for the model is the average of the variances for all points. To estimate the variance of a point and on N points of a model trained on M data sets, we compute:

$$\frac{1}{M}\sum(\bar{y}(x_n) - y_m(x_n)) \qquad var = \frac{1}{NM}\sum_{n=1}^{N}\sum_{m=1}^{M}(\bar{y}(x_n) - y_m(x_n))^2$$

where $\bar{y}(x_n)$ is the average of the predictions for point x_n. Figure 11.8 shows a model that overfits the data, which results in a large *variance*, showing that, for the point marked as a large circle, the predictions of individual hypotheses are spread in a broad range around their average.

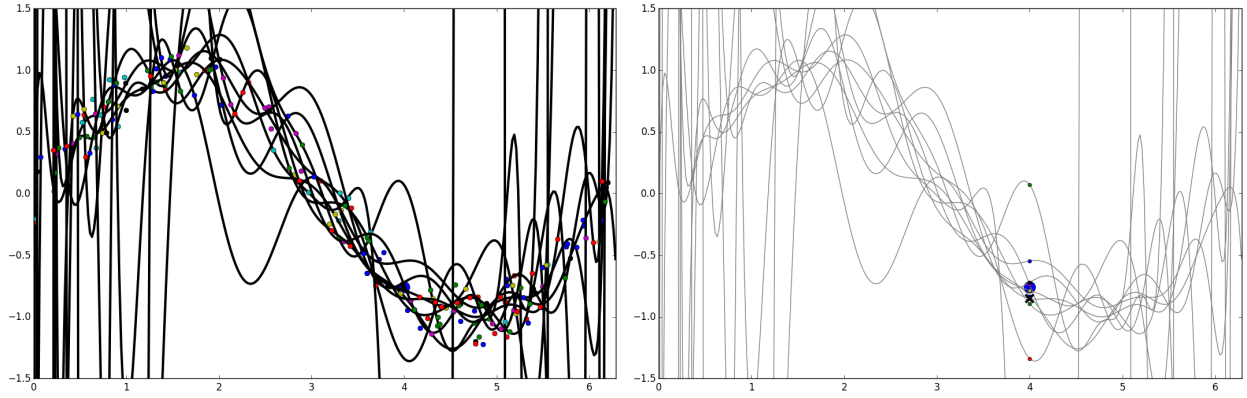

Figure 11.8: This model overfits the data and thus has a large *variance* in some points.

11.3 Bootstrapping

To estimate the *bias* and *variance* of a model we need to train the model over different training sets. However, in general we only have one training set, and so we need to resample our training set in order to generate different sets from the same distribution. One widely used resampling method is *bootstrapping*, which consists of creating replicas of the original set by sampling at random with reposition until we have a new set with the same number of points as the original. On average, the replica set will have around two thirds of the points of the original set, with some repetitions, leaving out about one third of the original points. With this method we can generate a large number of data sets and use them to estimate the *bias* and *variance* of our model.

The function below shows how we can create a number of replicas from a data matrix using bootstrapping. For each replica, the function generates a random vector with indexes of the rows of the data matrix to be copied to the replica. This random vector will contain repetitions (sampling with reposition), so some points will be left out. These are stored on the *test_sets* list of index vectors storing the points left out for each replica.

```
1 def bootstrap(replicas,data):
2     train_sets = np.zeros((replicas,data.shape[0],data.shape[1]))
3     test_sets = []
4     for replica in range(replicas):
5         ix = np.random.randint(data.shape[0],size=data.shape[0])
6         train_sets[replica,:] = data[ix,:]
7         in_train = np.unique(ix)
8         mask = np.ones(data.shape[0],dtype = bool)
9         mask[in_train] = False
10        test_sets.append(np.arange(data.shape[0])[mask])
11    return train_sets,test_sets
```

With the replicas, we can now estimate the bias and variance of a model. For this example, we'll use a polynomial regression model. We start by creating and filling a matrix with all the predictions of all polynomials fit to all the replicas of the training set. This is the `predicts` matrix in the source code below.

```
1 def bv_poly(degree, train_sets,test_sets, data):
2     replicas = train_sets.shape[0]
3     predicts = np.zeros((replicas,data.shape[0]))
4     for ix in range(replicas):
5         coefs = np.polyfit(train_sets[ix,:,0],
6                            train_sets[ix,:,1],degree)
7         predicts[ix,:] = np.polyval(coefs,data[:,0])
```

Then we count the number of times each point appears in the test sets and sum the predicted values and the squares of the predicted values. The squares are to compute the variance at each point, which can also be computed as:

$$var_n = \frac{1}{M}\sum_{m=1}^{M}\left(\bar{y}(x_n) - y_m(x_n)\right)^2 = y(\bar{x_n})^2 - y(\bar{x_n})^2$$

Where $y(\bar{x_n})^2$ is the average of the squares of the predictions and $y(\bar{x_n})^2$ is the square of the average of the predictions.

```
    counts = np.zeros(data.shape[0])
    sum_preds = np.zeros(data.shape[0])
    sum_squares = np.zeros(data.shape[0])
    for ix,test in enumerate(test_sets):
        counts[test] += 1
        preds = predicts[ix,test]
        sum_preds[test] = sum_preds[test]+preds
        sum_squares[test] = sum_squares[test]+preds**2
```

Finally, we compute the *bias* and *variance* for each point in each test set and average those values for all the points to estimate the *bias* and *variance*.

```
    var_point = (sum_squares - sum_preds**2/counts)/counts
    mean_point = sum_preds/counts
    bias = np.mean((data[:,-1]-mean_point)**2,axis=None)
    var = np.mean(var_point, axis = None)
    return bias,var
```

Since we are estimating *bias* and *variance* on each hypothesis with points that were not used to train that particular hypothesis, our estimates are unbiased. This is why it is important to distinguish between the points that were used to train each instance of the model (each hypothesis) and the points that were left out in each replica of our original data set.

11.4 Bias-variance decomposition

With a quadratic error function, the error is the expected square of the difference between the predicted values and the true values. This the loss function that is generally used in regression, so in regression we can decompose the error into:

$$E\left((y-t)^2\right) = \left(E(y)-E(t)\right)^2 + E\left(\left(y-E(y)\right)^2\right) + E\left(\left(t-E(t)\right)^2\right)$$

The term $\left(E(y)-E(t)\right)^2$ is the square of the difference between the expected prediction and the true value, which is the *bias*. $E\left(\left(y-E(y)\right)^2\right)$ is the *variance* and $E\left(\left(t-E(t)\right)^2\right)$ is the expected squared error between the expected value for each point and the value in the training set. This last term is the *noise* in our data set, which we will generally assume to be zero. Thus, assuming there is no random noise in our data, we can decompose the quadratic error into a sum of *bias* and *variance*. Figure 11.9 shows this decomposition used to examine the source of the error for polynomials of different degrees.

As we can see in Figure 11.9, there is a trade-off between *bias* and *variance*. If the model is underfitting, unable to adjust to the data, *bias* is the largest component of the error. But when overfitting, *variance* becomes the dominant factor. The optimal choice is the one that minimizes the total contribution of *bias* and *variance*.

So far, we've seen how to decompose the error into *bias* and *variance* for models using a quadratic error function. However, although this is the norm with regression problems, a quadratic error function is not ideal for classifiers. In these cases, we generally evaluate the error using a 0/1 loss function, giving an error of 1 if the predicted class is different from the true class, or 0 if they are equal. With this error function, the decomposition into *bias* and *variance* is different. First of all, the *main prediction*

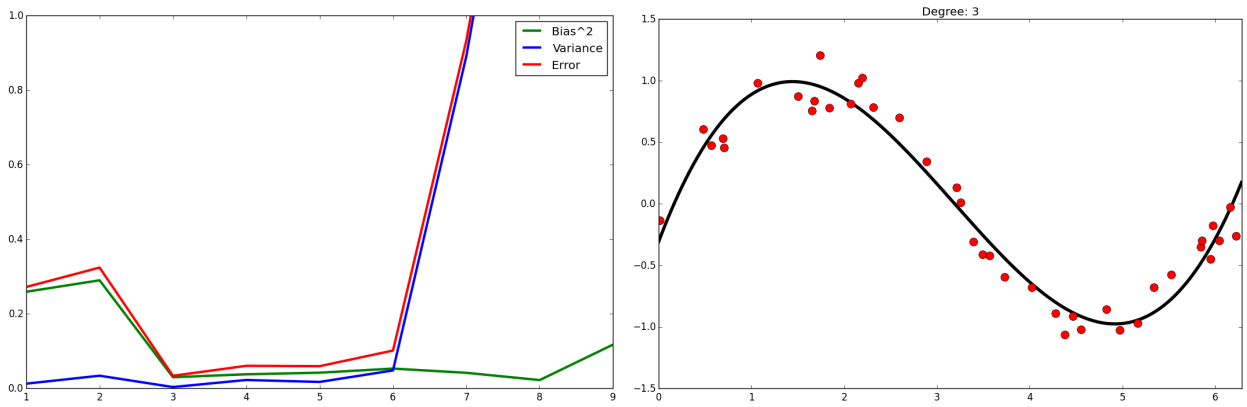

Figure 11.9: The left panel plots the *bias*, *variance* and total error (assuming zero noise). The right panel shows the result of training the best model.

in this case is the prediction that is most common, or the mode of the predictions, considering all hypotheses. So the *bias* for example i with a 0/1 loss function is the error of the *main prediction* with respect to the true class of point i:

$$bias_i = L(Mo(y_{i,m}), t_i)$$

where $Mo(y_{i,m})$ is the mode of predictions for point i over all m hypotheses and L is the loss function returning 0 if the values are equal or 1 if the values differ. The *variance* is the expected error of all predictions for example i with respect to the *main prediction*:

$$var_i = E\left(L(Mo(y_{i,m}), y_{i,m})\right)$$

So far, this is essentially the same as we saw for the quadratic error function used in regression problems. However, the error decomposition is fundamentally different because whether the *variance* increases or decreases the error depends on the *bias* for that error. If the *bias* is 0, meaning the *main prediction* is correct, then the *variance* increases the error, since any deviation from the main prediction increases the error. On the other hand, if the *bias* is 1, then this means the *main prediction* is incorrect and so any deviation from this prediction will decrease the expected total error. Thus, the error decomposition into *bias* and *variance* (assuming no noise in the data) is:

$$E\left(L(t, y)\right) = E\left(B(i)\right) + E\left(V_{unb.}(i)\right) \breve{} E\left(V_{biased}(i)\right)$$

where $V_{unb.}$ is the variance for points with *bias* of 0 and V_{biased} corresponds to the variance for points with *bias* of 1. Or, alternatively, we can consider the *variance* to be the net variance $E_x\left(V_{unb.}(i)\right) \breve{} E_x\left(V_{biased}(i)\right)$.

As an example, we'll decompose the *bias* and *variance* of a SVM classifiers (assuming the data has no noise). We start, as in the regression example, by fitting the classifier to each of the replicas and storing the predictions.

```
1 def bv_svm(gamma, C, train_sets, test_sets, data):
2     replicas = train_sets.shape[0]
3     predicts = np.zeros((replicas,data.shape[0]))
4     for ix in range(replicas):
5         sv = svm.SVC(kernel='rbf', gamma=gamma,C=C)
6         sv.fit(train_sets[ix,:,:-1],train_sets[ix,:,-1])
```

```
7         predicts[ix,:] = sv.predict(data[:,:-1])
```

Next, we count the occurrences of each point in the test sets and add the predictions, which are values of 0 or 1. The main prediction for each point is the more common prediction, computed by rounding the mean to 0 or 1. The *bias* is then computed from the difference between the main prediction and the true class, and the variance from the fraction of predictions that differ from the main prediction.

```
8     counts = np.zeros(data.shape[0])
9     sum_preds = np.zeros(data.shape[0])
10    for ix,test in enumerate(test_sets):
11        counts[test] += 1
12        sum_preds[test] = sum_preds[test]+predicts[ix,test]
13    main_preds = np.round(sum_preds/counts)
14    bias_point = np.abs(data[:,-1]-main_preds)
15    var_point = np.abs(sum_preds-counts*main_preds)/counts
```

Finally we average the *bias* and *variance* of each point over all the points to estimate the *bias* and *variance* of the model. However, we must distinguish the *variance* contributed by the unbiased points from the *variance* contributed by the biased points, returning the net variance.

```
16    u_var = np.sum(var_point[bias_point == 0])/float(data.shape[0])
17    b_var = np.sum(var_point[bias_point == 1])/float(data.shape[0])
18    bias = np.mean(bias_point)
19    return bias,u_var-b_var
```

Figure 11.10 shows the *bias* and *variance* decomposition of a SVM with a RBF kernel as a function of the γ parameter. Small γ make the radius of the RBF function larger and result in a classifier that effectively averages classes over a larger neighbourhood. This results in a larger *bias* and smaller *variance*. With larger values of γ, the *bias* decreases but the *variance* starts increasing. The right panel shows the classifier that best balances *bias* and *variance*, with $\gamma = 1$.

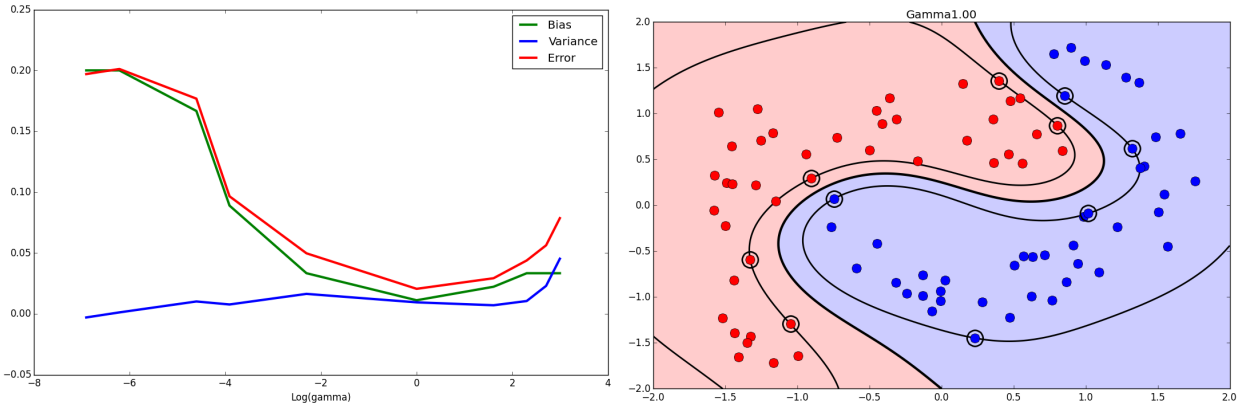

Figure 11.10: The left panel plots the *bias*, *variance* and total error (assuming zero noise) for different values of the logarithm of γ. The right panel shows the result of training the best model, with $\gamma = 1$.

11.5 Further Reading

1. Alpaydin [2], Section 4.3

2. Bishop [4], 4.1.2, 4.3.4, 7.1.3
3. (Optional: Valentini and Dietterich. Bias-variance analysis of support vector machines for the development of SVM-based ensemble methods [21])
4. (Optional: Domingos, P. A unified bias-variance decomposition [8])

Chapter 12

Ensemble Methods

Ensemble methods. Bagging and bragging. Boosting and stumping.

12.1 Bagging

Ensemble methods combine different hypotheses, whether from regression models or classifiers, in order to improve prediction. One way of doing this is to train different instances of some model with different training sets and then aggregate the response, either with an average, for regression problems, or with majority voting for classification. We can obtain replicas of the training set with bootstrapping, as we saw on Chapter 11, and use those to train different hypotheses. In the code below, we first create a vector pxof x values to plot our polynomial curves and a matrix for all the predictions. Then we fit the polynomial model to each replica and compute the predicted values. Finally, we compute the mean of each prediction. This method is called *bootstrap aggregating* or *bagging*, for short.

```
train_sets, _ = bootstrap(replicas,data)
px = np.linspace(ax_lims[0],ax_lims[1],points)
preds = np.zeros((replicas,points))
for ix in range(replicas):
    coefs = np.polyfit(train_sets[ix,:,0], train_sets[ix,:,1],degree)
    preds[ix,:] = np.polyval(coefs,px)
mean = np.mean(preds,axis=0).ravel()
```

Alternatively, we can also use the median instead of the mean for the ensemble. This variant of *bagging* is called *bragging*. Figure 12.1 compares these two variants on a polynomial regression problem. The ensemble of curves was computed using the replicas obtained by bootstrapping.

For classification with *bagging*, instead of averaging the predicted value, the class is predicted by majority voting among the classifiers in the ensemble. Apart from this, the procedure is identical: train the model on a number of replicas of the training set, obtained by bootstrapping. Then apply each resulting classifier and classify according to the class that was given the most times. The code below shows how to create an ensemble of SVM classifiers and then use it to classify new points.

```
train_sets,_ = bootstrap(replicas,data)
gamma = 2
C=10000
svs = []
```

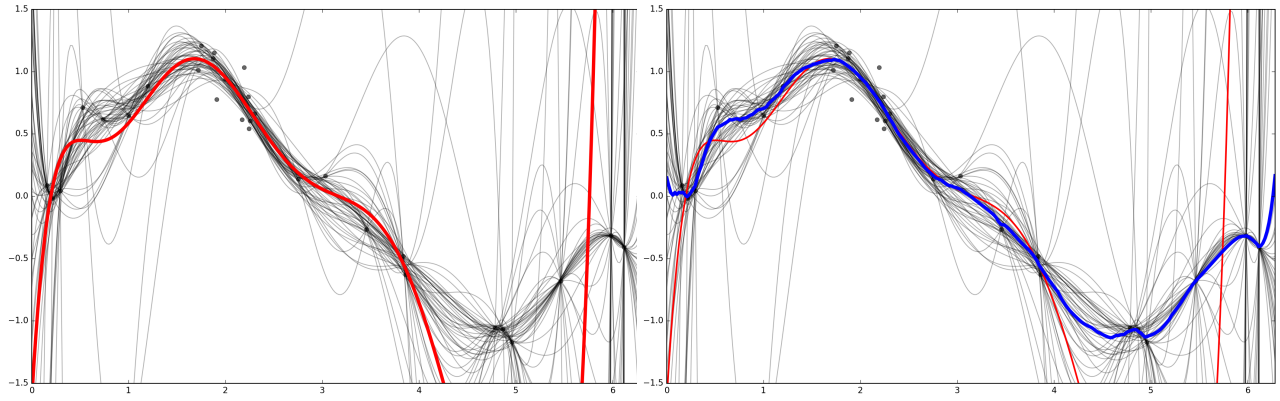

Figure 12.1: The two panels show the instances obtained by training the polynomial model with different replicas of the data set. The left panel shows the ensemble predictions using the mean (*bagging*). The right panel also shows the result using the median (*bragging*).

```
5  pX,pY = np.meshgrid(pxs,pys)
6  pZ = np.zeros((len(pxs),len(pys)))
7  for ix in range(replicas):
8      sv = svm.SVC(kernel='rbf', gamma=gamma,C=C)
9      sv.fit(train_sets[ix,:,:-1],train_sets[ix,:,-1])
10     svs.append(sv)
11     preds = sv.predict(np.c_[pX.ravel(),pY.ravel()]).reshape(pZ.shape)
12     pZ = pZ + preds
13 pZ = np.round(pZ/float(replicas))
```

The `meshgrid` call on line 5 generates the `pX` and `pY` matrices for plotting the contour, along with the `pZ` matrix with the prediction values. Then, for each replica, we train a new SVM classifier and add its predictions to `pZ`. The majority class, 0 or 1, is computed by rounding the average classification. Figure 12.2 shows the 50 SVM classifiers computed from the replicas of the data-set and the result of the ensemble classifier, using the majority vote from the 50 SVM to classify each point.

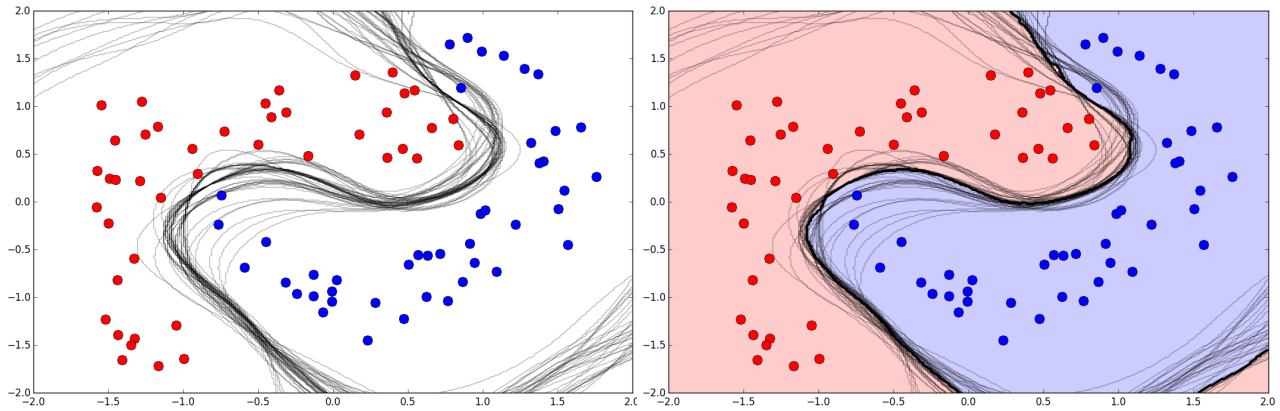

Figure 12.2: The two panels show the 50 SVM classifiers and the resulting ensemble classifier.

Bootstrap aggregating reduces variance without increasing bias and so it is useful if the base model has a high variance and low bias. In classification, the probability of the ensemble classifying an example correctly increases rapidly with the number of classifiers aggregated. For an ensemble of T classifiers, each with a probability p of correctly classifying an example, the probability of a correct classification with majority voting is:

$$\sum_{k=T/2+1}^{T} \binom{T}{k} p^k (1-p)^{T-k}$$

Figure 12.3 shows this increase for different values of p. Even modest classifiers can become quite accurate if aggregated in large ensembles.

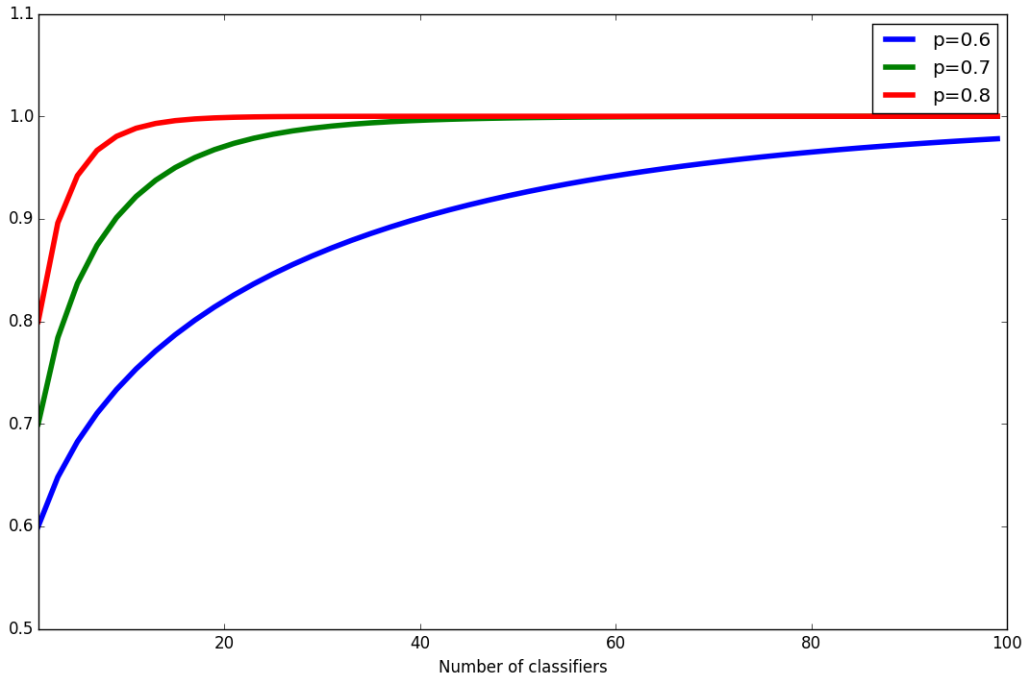

Figure 12.3: Probability of correct classification for different values of p as a function of the number of classifiers in the *bagging* ensemble.

However, this increase in probability presumes that the classifiers are statistically independent. The more correlation there is between classifiers the smaller the improvement gained by the ensemble. This is why it is important to use *unstable classifiers* with a high variance, otherwise they will all be similar and there will be no advantage to aggregating them. *Unstable classifiers* are classifiers that are sensitive to changes in the training set. *Stable classifiers*, such as SVM with a small regularization constant (C) or k-NN, are not good choices for *bagging*.

12.2 Boosting

For *boosting* we'll consider the *adaptive boosting* algorithm, or *AdaBoost*. This method finds a linear combination of weak classifiers, where each individual classifier is assigned a weight, so that the weighted average of the classifier responses minimizes the classification error. This is done by training each classifier with the same data set but giving different weights to different data points, weighing more strongly those that were previously misclassified.

So, first, we initialize the weights of all N examples to $w_n = 1/N$. Then we train one classifier so that we minimize the weighted error of the training set:

$$J_m = \sum_{n=1}^{N} w_m^n I(y_m(x^n) \neq t^n)$$

Then we compute the weighted error of the classifier and the weight of the classifier in the ensemble classifier.

$$\epsilon_m = \frac{\sum_{n=1}^{N} w_m^n I(y_m(x^n) \neq t^n)}{\sum_{n=1}^{N} w_m^n} \qquad \alpha_m = \ln \frac{1-\epsilon_m}{\epsilon_m}$$

Then we use α_m to update the weights for the data points [1]:

$$w_{m+1}^n = w_m^n \exp\left(\alpha_m I(y_m(x^n) \neq t^n)\right)$$

The indicator function I returns 1 if the values are different, 0 if they are equal, so this results in increasing the weights of misclassified points. Then a new classifier is fitted to the training set with the updated weights, and the process is repeated until the weighted training error is zero or greater than 0.5. To use the final ensemble classifier obtained with *AdaBoost* we compute the weighted sum of the responses of the individual classifiers: $f(x) = sign \sum_{m=1}^{M} \alpha_m y_m(x)$

For an example, we'll use a *decision stump* as the weak classifier. This is a decision tree with only one level, and consists of finding the threshold value of a single feature in the data that best splits the classes we wish to separate. For a data set with two dimensions, this means we are creating horizontal and vertical lines to split the data. *Adaptive boosting* with *decision stumps* is called *stumping*.

At each iteration of the *AdaBoost* algorithm, we fit a decision tree classifier with depth of 1 to the weighted data set (line 7), compute the weighted error (lines 8 and 9) and update the weights, storing the value of α_m, computed in line 10. Line 12 normalizes the weights so that they add up to 1.

```
1 from sklearn.tree import DecisionTreeClassifier
2 hyps = []
3 hyp_ws = []
4 point_ws = np.ones(data.shape[0])/float(data.shape[0])
5 max_hyp = 50
6 for ix in range(max_hyp):
7     stump = DecisionTreeClassifier(max_depth=1)
8     stump.fit(data[:,:-1], data[:,-1], sample_weight = point_ws)
9     pred = stump.predict(data[:,:-1])
10    errs = (pred != data[:,-1]).astype(int)
11    err = np.sum(errs*point_ws)
12    alpha = np.log((1-err)/err)
13    point_ws = point_ws*np.exp(alpha*errs)
14    point_ws = point_ws/np.sum(point_ws)
```

Figure 12.4 shows the first three iterations and the final classifier, after 10 iterations. The points are shown in different sizes depending on their current weights.

To classify new points and compute the error, we iterate through the stored classifiers, compute the prediction of each (line 3) and add it, weighted by the respective weight of that classifier (line 4).

```
1 net_pred = np.zeros(data.shape[0])
2 for ix in range(len(hyps)):
```

[1] In the original algorithm, by Freund and Schapire, it was $\alpha_m = \frac{1}{2} \ln \frac{1-\epsilon_m}{\epsilon_m}$. However, this $\frac{1}{2}\alpha$ just affects the weights of the classifier by a constant and the weights of the points by a value that is independent of the point, so can be omitted

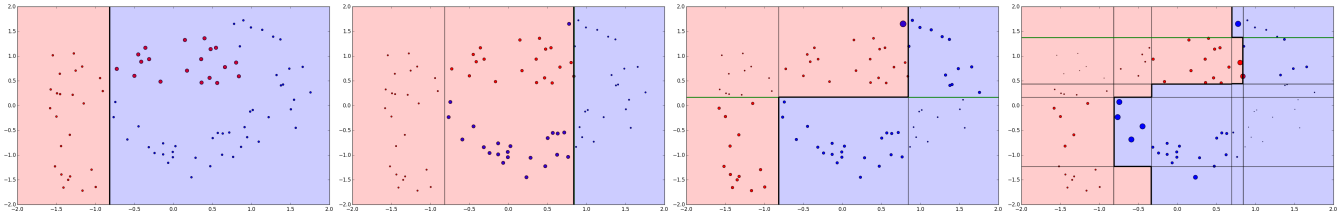

Figure 12.4: Adaptive boosting using decision stumps.

```
    pred_n = hyps[ix].predict(data[:,:-1])
    preds = preds+pred_n*hyp_ws[ix]
net_pred[preds<0] = -1
net_pred[preds>=0] = 1
errors = np.sum((net_pred !=data[:,-1]).astype(int))
```

AdaBoost can be seen as a sequential minimization of an exponential function of the weighted error:

$$E = \sum_{n=1}^{N} \exp\left(-t_n f_m(x_n)\right) \qquad f(x) = \sum_{m=1}^{M} \alpha_m y_m(x)$$

It is sequential because, at each step m, we assume the classifiers and weights for all steps $1, \ldots, m-1$ are fixed, and so the error function at each step is simply:

$$E = \sum_{n=1}^{N} w_m^n \exp\left(-t_n \alpha_m y_m(x_n)\right)$$

which we can decompose by separating the terms corresponding to points that are correctly classified, with $t_n = y_m(x_n)$, and those that are not correcly classified, with $t_n \neq y_m(x_n)$. Letting set $\mathcal{T}$ be the set of points correctly classified by classifier m and set $\mathcal{M}$ the set of points misclassified by classifier m, the error function is:

$$E = e^{-\alpha_m} \sum_{n \in \mathcal{T}} w_m^n + e^{\alpha_m} \sum_{n \in \mathcal{M}} w_m^n = e^{-\alpha_m} \sum_{n=1}^{N} w_m^n + (e^{\alpha_m} - e^{-\alpha_m}) \sum_{n=1}^{N} w_m^n I(y_m(x^n) \neq t^n)$$

where function I is the indicator function that returns 1 when the point is misclassified or 0 if it is classified correctly. Minimizing as a function of y_m and α_m, we obtain the following solutions:

$$J_m = \sum_{n=1}^{N} w_m^n I(y_m(x^n) \neq t^n) \qquad \alpha_{m+1} = \ln \frac{1 - \epsilon_m}{\epsilon_m}$$

which correspond to the weighted error function for *AdaBoost* and the expression for computing α_m.

12.3 Further Reading

1. Alpaydin [2], Sections 17.6 and 17.7
2. Marsland [17], Chapter 7

3. Bishop [4], 14.2, 14.3

Part II

Learning Theory

Chapter 13

Probably Approximately Correct Learning

Empirical Risk Minimization. Decision theory. Probably Approximately Correct Learning. VC dimension and shattering.

In Chapter 11 we saw how there is a trade-off between the ability of a model to fit the training data and the ability of the model to generalize from the training sample to the population of examples whose features we wish to predict. We did this by instantiating the model into different hypotheses, using different training sets (by *Bootstrapping*) and then measuring the *Bias*, which is the error of the mean prediction for each example, and the *Variance*, the dispersion of the predictions for each example. We saw how reducing *Bias* leads to an eventual increase in *Variance* due to *overfitting*. In this chapter we will look at the *Bias-Variance tradeoff* in more detail, with a more formal and grounded approach.

13.1 Empirical Risk Minimization

In brief, *Empirical Risk Minimization* consists in minimizing the training error. Or, more generally, minimizing a loss function measured on the training set, such as the classification error or the quadratic error in regression. This is what we have been doing when training regression or classification models in supervised learning. The name comes from trying to minimize the risk, which is the expected loss, and this is an empirical risk because we measure it on the training set. This contrasts with the true risk, or the average loss over all possible data, which we cannot measure directly. Furthermore, if we adjust the parameters to minimize the empirical risk, then the empirical risk becomes a biased estimate of the true risk (for example, the true error, if that is our loss function). However, we can use probability theory to find a probable upper bound on the true error based on the empirical error we minimized.

First, we note that, if $A_1, A_2, ..., A_k$ are random events, then the probability of at least one of them occurring cannot be larger than the sum of their probabilities:

$$P(A_1 \cup A_2 \cup ...A_k) \leq P(A_1) + P(A_2) + ... + P(A_k)$$

This is the *union bound*, an upper bound on the probability of the union of a set of random events. Furthermore, if $B_1, B_2, ..., B_m$ are independent random events following the Bernoulli distribution, which is the distribution of a random variable that can take the values 0 or 1 with probabilities ϕ and

$1 - \phi$ respectively, with $\hat{\phi}$ defined as:

$$P(B_i = 1) = \phi \qquad \hat{\phi} = \frac{1}{m} \sum_{i=1}^{m} B_i$$

Then, the following *Hoeffding's inequalities* hold:

$$P(\phi - \hat{\phi} > \gamma) \leq e^{-2\gamma^2 m}$$

$$P(\hat{\phi} - \phi > \gamma) \leq e^{-2\gamma^2 m}$$

In other words, the probability that the mean of a set of random Bernoulli variables with the same probability $P(B_i = 1) = \phi$ deviating from ϕ by more than γ decreases exponentially with γ and the number of examples on the sample.We can rewrite this as the *Hoeffding's inequality*:

$$P(|\phi - \hat{\phi}| > \gamma) \leq 2e^{-2\gamma^2 m}$$

This is useful because, in classification, we can consider the classification error for each example to be a Bernoulli random variable, with values of 0 or 1, and ϕ to be the probability of the classifier committing an error. In this case, $\hat{\phi}$ is the observed error rate on the training set, or the *empirical error*, and we train the classifier by finding the set of parameters that minimizes this error. Thus, we are doing *empirical risk minimization*(ERM) because the empirical error, which we try to minimize, is the risk of misclassification for examples in the training set. However, what we would really like would be to minimize the ϕ, the true error, which we cannot measure but is related to the empirical error $\hat{\phi}$ by Hoeffding's inequality. This gives us a probable upper bound on the true error and is the rationale behind the notion of *Probably Approximately Correct Learning*.

13.2 Probably Approximately Correct Learning

Let us consider $\mathcal{X}$ be the population of all possible examples and $c : \mathcal{X} \to \{0, 1\}$ the target function to learn, assigning each example to one of two possible classes. $\mathcal{H}$ is the hypothesis class the learner will explore and $\mathcal{D}$ is the probability distribution according to which examples are drawn from $\mathcal{X}$, and according to which the training sample S is obtained. Our learner will draw S from $\mathcal{X}$ according to distribution $\mathcal{D}$ and then find an hypothesis $\hat{h}$ that minimizes the empirical error, $\hat{E}_s$, measured on the sample S:

$$\hat{h} = \arg\min_{h \in \mathcal{H}} \hat{E}_S(h)$$

This is the empirical error, which is the average error on the sample, while the true error of an hypothesis h is the probability of error for any example drawn from $\mathcal{X}$ according to distribution $\mathcal{D}$. In other words, the true error corresponds to the set of possible instances for which the learned hypothesis differs from the target function c:

$$E(h) = P_{x \sim D}\left(h(x) \neq c(x)\right)$$

The true error is not accessible to the learner, who can only compute the empirical error.

In general, it is not reasonable to assume that the true error will be zero, since we cannot include all possible examples in the training set and different hypotheses may seem correct on all the training set while making mistakes outside it. So we need a more realistic set of requirements for our learner. We

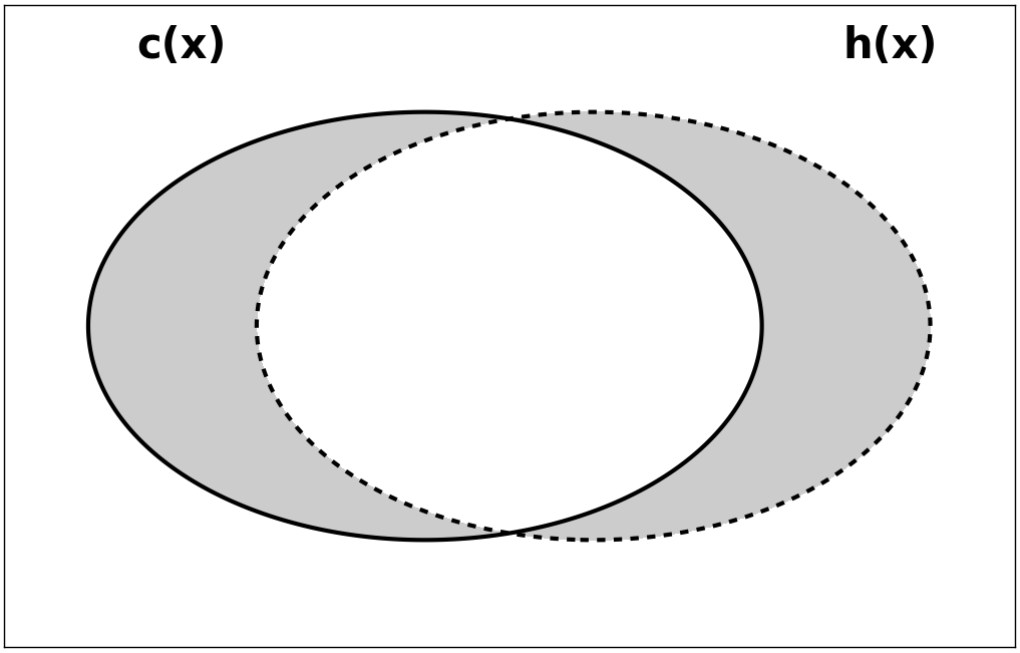

Figure 13.1: The true error of an hypothesis is the difference between the classifications given by that hypothesis and the classifications given by the function c providing the true classes of all points.

can demand that the result is *approximately correct*, in the sense that the true error of the hypothesis we find be below some threshold ϵ, instead of zero:

$$E(\hat{h}) \leq \epsilon$$

Furthermore, since we are training our classifier on a random subset of all possible examples, the training set may mislead our classifier into finding a hypothesis whose true error is not even bound by ϵ. So we require that our learner is *Probably Approximately Correct* (PAC):

$$P\left(E(\hat{h}) \leq \epsilon\right) \geq 1 - \delta$$

with $\epsilon < 1/2$ and $\delta < 1/2$. That is, there is a probability $1 - \delta$, with a small (below 0.5) δ, that the true error of the resulting hypothesis is some ϵ below 0.5. A learner is an *Efficient PAC learner* if it can learn hypothesis $\hat{h}$ in a time that is polynomial on $1/\epsilon$ and $1/\delta$.

Let us now suppose that we have a PAC learner, able to learn an hypothesis with a true error of ϵ or less with a probability of $1 - \delta$ or more, and let us assume that the hypothesis space $\mathcal{H}$ is finite and contains at least one hypothesis with $E(h) \leq \epsilon$, which must be true for there to be a chance of finding such hypotheses. Training and testing examples will all be drawn from $\mathcal{X}$ according to distribution $\sim \mathcal{D}$. Let us also define the *version space* as the set of *consistent hypotheses*, which are those hypotheses for which the empirical error is zero. This means that any consistent hypothesis (any hypothesis in the version space) minimizes the empirical error, since the empirical error cannot be less than zero.

We say that the version space is ϵ*-exhausted* if all hypotheses in the version space have a true error of at most ϵ:

$$\forall h \in \mathcal{V} \qquad E(h) < \epsilon$$

It is important to note that the learner cannot tell this, since the true error is not measurable by the learner. Conversely, the version space is not ϵ*-exhausted* if at least one hypothesis has a true error greater than ϵ.

What is the probability that no hypothesis in the version space has a true error larger than ϵ? In other words, what is the probability that the version space is ϵ-exhausted? If we suppose $h_1, h_2, ..., h_k$ are hypotheses with a true error greater than ϵ, $E(h_i) > \epsilon$, then the probability that h_i is consistent with one example is smaller than $1 - \epsilon$, since that is the probability of correct classification for a hypothesis with error ϵ. Thus, the probability of the hypothesis being consistent with all examples in a set of m examples is below $(1 - \epsilon)^m$. Using the *union bound* relation we saw previously:

$$P(A_1 \cup A_2 \cup ...A_k) \leq P(A_1) + P(A_2) + ... + P(A_k)$$

we know that the probability that any hypothesis h_i of the k hypotheses with $E(h_i) > \epsilon$ is consistent with m examples is $\leq k(1 - \epsilon)^m$, which is the sum of the probabilities of each hypothesis h_i being consistent with the set of examples. Although we do not know the value of k, which is the total number of such hypotheses, we know that k cannot be larger than the total number of hypotheses, $|\mathcal{H}|$. That is, $k(1 - \epsilon)^m \leq |\mathcal{H}|(1 - \epsilon)^m$. And since $(1 - \epsilon) \leq e^{-\epsilon}$ for $0 < \epsilon < 1$, the probability of an hypothesis with a true error greater than ϵ being in the version space (that is, being compatible with the training set) is bounded by:

$$P\left(\exists h \in \mathcal{V} : E(h) \geq \epsilon\right) \leq |\mathcal{H}|e^{-\epsilon m}$$

Let us now choose a value δ that is an upper bound on the probability that an hypothesis in the version space has a true error greater than ϵ. In this case, for $P\left(E(h \in \mathcal{V}) > \epsilon\right) \leq \delta$,

$$|\mathcal{H}|e^{-\epsilon m} \leq \delta \Leftrightarrow m \geq \frac{1}{\epsilon}\left(\ln \frac{|\mathcal{H}|}{\delta}\right)$$

This gives us a lower bound on the number of examples needed to have a probability of at least $1 - \delta$ of learning an hypothesis with a true error of at most ϵ.We can also compute the lower bound on ϵ as a function of the size of the training set, m, and the probability δ that the learner produces an hypothesis with an error greater than ϵ:

$$m \geq \frac{1}{\epsilon}\left(\ln \frac{|\mathcal{H}|}{\delta}\right) \Leftrightarrow \epsilon \leq \frac{1}{m}\left(\ln \frac{|\mathcal{H}|}{\delta}\right)$$

This assumes that the learner is a *consistent learner*. That is, a learner that learns hypothesis with zero empirical error, $\hat{E_S}(\hat{h}) = 0$. To extend this reasoning for $\hat{E_S} \geq 0$, we can consider the empirical (training) error to be the mean of Bernoulli variables corresponding to the classification error of each training example:

$$\hat{E}(h_i) = \frac{1}{m}\sum_{i=1}^{m} 1\{h(x^{(i)} \neq c(x^{(i)})\} = \frac{1}{m}\sum_{i=1}^{m} Z_i$$

Applying the Hoeffding inequalities we saw before:

$$P(\phi - \hat{\phi} > \gamma) \leq e^{-2\gamma^2 m}$$

$$P(\hat{\phi} - \phi > \gamma) \leq e^{-2\gamma^2 m}$$

gives us the following bounds:

$$P\left(\hat{E} - E > \epsilon\right) \leq e^{-2m\epsilon^2}$$

$$P\left(E - \hat{E} > \epsilon\right) \leq e^{-2m\epsilon^2}$$

Thus, the probability of the true error of hypothesis h being more than ϵ above the empirical error of h is bounded by:

$$P\left(E(h) > \hat{E}_S(h) + \epsilon\right) \leq e^{-2m\epsilon^2}$$

Extending this for all hypotheses $h \in \mathcal{H}$:

$$P\left(\exists h \in \mathcal{H} : E(h) > \hat{E}_S(h) + \epsilon\right) \leq |\mathcal{H}|e^{-2m\epsilon^2}$$

Calling this probability δ and solving for m, we obtain:

$$m \geq \frac{1}{2\epsilon^2}(\ln \frac{|\mathcal{H}|}{\delta})$$

This gives us the lower bound on the size of the training set to guarantee a maximum probability of $delta$ that the true error of the hypothesis we find is greater than the sum of the empirical error and ϵ. This lower bound increases with the square of $1/\epsilon$ and with the logarithm of the total number of hypotheses, $|\mathcal{H}|$.

This result gives us an important insight into a major problem of machine learning, which is the *inductive bias*. We mentioned before that it is always necessary to assume something about the hypothesis space we are learning in order to be able to generalize from the training set to future examples. This assumption that restricts the hypothesis space is the inductive bias. For example, that the best regression curve will be a polynomial of some degree or the the best classifier will be a hyperplane with a specified number of dimensions. Let us now look at what happens with a classifier that has no inductive bias. For example, suppose that our hypothesis space $\mathcal{H}$ is the set of all subsets of $\mathcal{X}$. This means that our classifier can split $\mathcal{X}$ into two classes in any combination of examples by finding a subset of $\mathcal{X}$ defining one class and placing in the other class any example not in that subset. If this is the case, then the size of our hypothesis space is two raised to the number of possible examples, since each example may or may not belong to each subset: $|\mathcal{H}| = 2^{|\mathcal{X}|}$. Let us further assume that each example is described by a vector of n boolean features, for simplification, which means that $\mathcal{X}$ is the set of all 2^n combinations of features and the cardinality of our hypothesis space is:

$$|\mathcal{H}| = 2^{|\mathcal{X}|} = 2^{2^n}$$

Using this, we can compute the lower bound on the size of the training set for some value of ϵ and δ:

$$m \geq \frac{1}{2\epsilon^2}(\ln \frac{|\mathcal{H}|}{\delta}) \Leftrightarrow m \geq \frac{1}{2\epsilon^2}(2^n \ln \frac{2}{\delta})$$

The lower bound of m grows exponentially in the number of features, n, and since for an approximately correct learning we want ϵ to be below 0.5, this means that m will be greater than $|\mathcal{X}|$, the total number of all possible examples. In other words, without inductive bias we have no probably approximately correct learning when trying to extrapolate from the training set to new examples.

Let us now extend this analysis to the hypotheses obtained by empirical risk minimization (ERM). Recall that an ERM learner selects the hypothesis from $\mathcal{H}$ that minimizes the empirical error:

$$\hat{h} = \arg\min_{h \in \mathcal{H}} \hat{E}(h)$$

Let us define the *generalization error* as the difference between the true error and the empirical error:

$$E(\hat{h}) - \hat{E}(\hat{h})$$

and h^* be the best hypothesis, in the sense of being the hypothesis with the smallest true error.

$$h^* = \arg\min_{h \in \mathcal{H}} E(h)$$

Let $1 - \delta$ be the probability that the true error of the ERM hypothesis is not greater than the empirical error plusϵ, $P(E(\hat{h}) \leq \hat{E}(\hat{h}) + \epsilon) = 1 - \delta$. Furthermore, the empirical error for the ERM hypothesis, given our training set S, cannot be greater than the empirical error of the best hypothesis, since the ERM hypothesis was obtained by minimizing the empirical error. Thus, $\hat{E}(\hat{h}) \leq \hat{E}(h^*)$. This means that the true error of the best hypothesis must also be bounded by the sum of the empirical error of the best hypothesis and ϵ with a probability of at least $1 - \delta$, because the best hypothesis, by definition, is the hypothesis with the lowest true error: $\hat{E}(h^*) \leq E(h^*) + \epsilon$ with $P \geq 1 - \delta$.

Combining all these, we find that, with a probability of at least $1 - \delta$, the true error of the ERM hypothesis we obtain by minimizing the empirical error cannot be greater than the true error of the best hypothesis plus two times ϵ: and $P(E(\hat{h}) \leq E(h^*) + 2\epsilon) \geq 1 - \delta$. Using the previous bounds, we can decompose the true error of the ERM hypothesis into these two terms:

$$E(\hat{h}) = \left(\arg\min_{h \in \mathcal{H}} E(h) \right) + 2\sqrt{\frac{1}{2m} \ln \frac{|\mathcal{H}|}{\delta}}$$

The first term is the smallest true error of any hypothesis in the hypothesis space $\mathcal{H}$, which corresponds to the *Bias* of our model, and the larger this term, the less the model is able to fit the data adequately. Thus, when this term dominates the true error, we say that our model is underfitting. The second term is a function of the size of the hypothesis space and the size of the training set, and corresponds to the *Variance* of our model. In general, the larger the hypothesis space the greater the variance of the predictions of the hypotheses obtained by training with different training sets. If this term dominates the true error, the model is overfitting since now the critical problem is not the model's inability to adjust to the points but rather its excessive freedom in adapting to the training set.

13.3 Shattering and the V-C Dimension

So far, we have assumed that the hypothesis space $\mathcal{H}$ is finite, which allowed us to obtain a lower bound for the size of the training set given the values of ϵ and δ:

$$m \geq \frac{1}{2\epsilon^2} \left(\ln \frac{|\mathcal{H}|}{\delta} \right)$$

This can be true for some classifiers, such as decision trees with a fixed limited depth, but is not true in general, as it is often the case that classifiers use continuous parameters and thus have an infinite number of possible hypotheses. For example, logistic regression, SVM, neural networks and so forth. In these cases, the previous expression is no longer useful and we need another approach.

We can start by thinking that, for a classifier with continuous parameters, there can be many hypotheses that result in the same set of labels for a given set of examples. A logistic regression, for example, can divide the same set of points into the same two subsets with an infinitude of lines, as long as the lines are placed between the sets to separate, as illustrated in Figure 13.2.

So what is relevant is how a classifier divides the set of examples into different subsets and not how many different decision frontiers it can express. This leads us to the following definition:

Hypothesis class $\mathcal{H}$ shatters set of points $\mathcal{S}$ if, for any labelling S of $\mathcal{S}$, there is a $h \in \mathcal{H}$ that is consistent with S

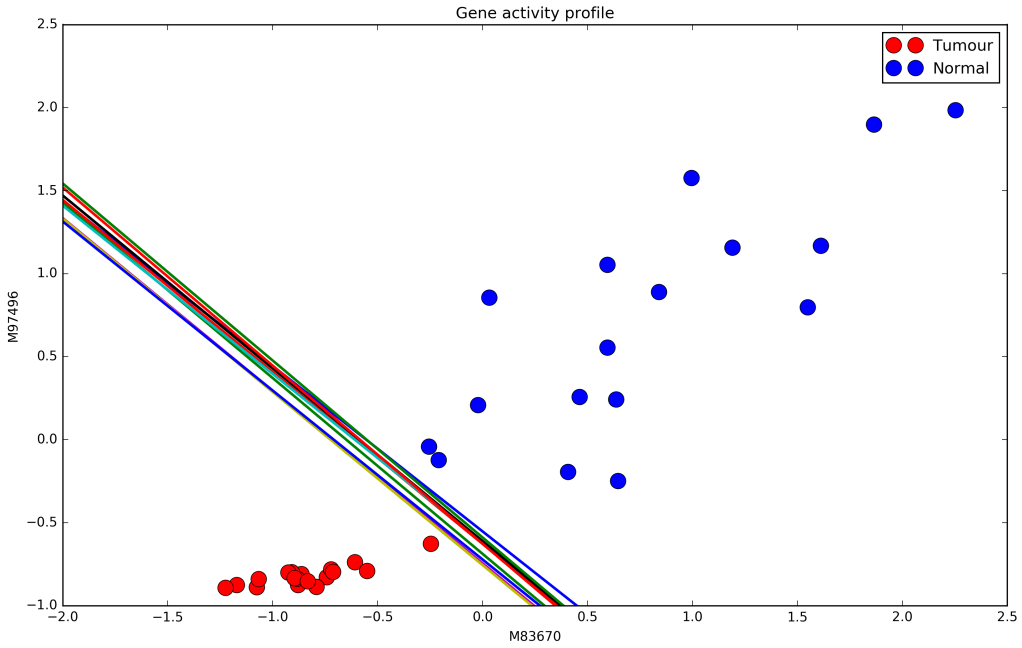

Figure 13.2: For classifiers with continuous parameters, although the size of the hypothesis space is infinite, there are also infinite hypothesis resulting in the same classifications for all points.

In other words, $\mathcal{H}$ shatters a set of points if it can provide hypotheses that can classify all those points correctly whatever the class each point belongs to. For example, a linear classifier in two dimensions can shatter a set of 3 points forming a triangle, as shown in Figure 13.3.

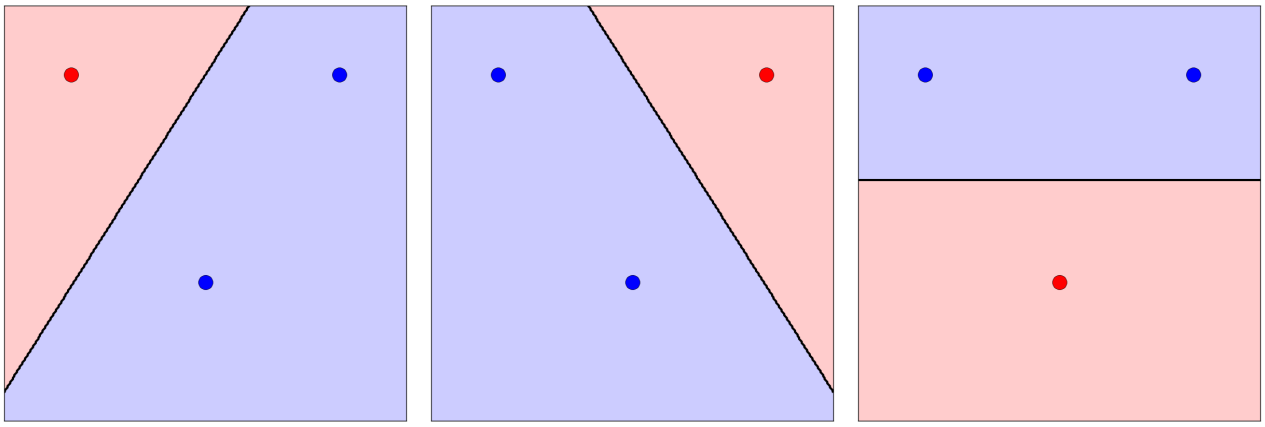

Figure 13.3: A linear classifier in two dimensions can shatter this set of 3 points by correctly classifying them whatever their labels. Note that two other cases, where all points belong to the same class, were omitted for being trivial.

Using this notion of *shattering*, we can define the *Vapnik-Chervonenkis dimension* of an hypothesis space $\mathcal{H}$, or, for short, the *V-C dimension* of $\mathcal{H}$, $VC(\mathcal{H})$, as the size of the largest set of points that $\mathcal{H}$ can shatter. Note that the points can be placed in the most adequate way to facilitate shattering and that there may be sets with fewer than $VC(\mathcal{H})$ points that $\mathcal{H}$ cannot shatter. For example, if two points overlap in the same coordinates no hypothesis can distinguish them. What matters is that some set of points exists for which the hypothesis space can provide hypotheses for correct classification whatever

the labels may be. Vapnik et. al. demonstrated that, with a probability of $1 - \delta$:

$$E(\hat{h}) \leq \hat{E}(\hat{h}) + \mathcal{O}\left(\sqrt{\frac{VC(\mathcal{H})}{m} \ln \frac{m}{VC(\mathcal{H})} + \frac{1}{m} \ln \frac{1}{\delta}}\right)$$

That is, the true error of the ERM hypothesis is bounded by the empirical error plus a term that is approximately proportional to the VC dimension of the hypothesis space ($VC(\mathcal{H})$) and approximately inversely proportional to the size of the training set (m). In other words, to keep the true error within some bounds, the size of the training set must increase as $VC(\mathcal{H})$ increases.

This has implications for the approach of using linear discriminants in higher dimensions to classify non linearly separable sets. The VC dimension of a linear classifier is $D + 1$, where D is the dimension of the feature vectors. As we increase the dimension D, we increase the VC dimension of the hypothesis space and thus we require a larger sample for the training set to prevent overfitting and an increase in the generalization error.

13.4 Summary

The probabilistic and statistical foundation of machine learning provides us with a good intuition about important aspects, even though, in practice, methods such as validation and testing provide better estimates of the true error of our models or hypotheses. In this chapter we saw how inductive bias is an important requirement for machine learning, since without it the hypothesis space becomes too large for allowing generalization from a data set to all possible points. We also saw how the true error results from a contribution of the error of the best hypothesis, corresponding to the bias of the model, and the generalization error due to the size of the hypothesis space, corresponding to the variance of the model. This is the source of the Bias-Variance tradeoff, since improving the best hypothesis of the model generally requires increasing the size of the hypothesis space. Most importantly, we saw the notion of Probably Approximately Correct learning. In machine learning we cannot guarantee that the prediction error will be zero but we can make it probable that it will be small, as long as we have enough data.

13.5 Further Reading

1. Alpaydin [2], Sections 2.1 through 2.3
2. Mitchell [18], Chapter 7 up to section 7.4

Chapter 14

Decision

Bayesian Decision theory. Maximum a posteriori estimation. Decisions and costs.

14.1 Bayesian Decision theory

So far, we saw several examples of maximizing likelihood as a way to find the best parameters to fit some set of labelled data. The maximum likelihood (ML) approach consists of maximizing the predicted joint probability of all features and labels, $p(x, y)$, by adjusting θ, which is the vector of parameters. Since the joint probability $p(x, y)$ can be decomposed into the conditional probability of y given x multiplied by the marginal probability$p(x)$, and since $p(x)$ is independent of the parameters of our model, θ, we can simplify the maximization problem:

$$\begin{aligned}\hat{\theta}_{ML} &= \arg\max_{\theta} \prod_{t=1}^{n} p(x^t, y^t; \theta) \\ &= \arg\max_{\theta} \prod_{t=1}^{n} p(y^t|x^t; \theta) \times \prod_{t=1}^{n} p(x^t) \\ &= \arg\max_{\theta} \prod_{t=1}^{n} p(y^t|x^t; \theta)\end{aligned} \tag{14.1}$$

In other words, we choose the parameters that maximize the probability of the observed labels given the observed features. These are the maximum likelihood parameters. Note that, in this case, the probabilities are a function of the parameters but the parameters, θ, are not considered to be random variables. This is because, under a frequentist interpretation, probability is the measure of the frequency of a random event in the limit of infinite trials and the parameters values are not random events. Thus, in a frequentist interpretation, it does not make sense to consider the probability of the parameters having those values or the probability distribution of the parameters.

However, under a Bayesian interpretation, probability is a measure of rational confidence about some value or outcome and a probability distribution describes our uncertainty about the values. Under this interpretation, θ can be considered to be just another random variable, like the features x or the labels y. Thus, under a Bayesian approach, we may want to find the most probable values of θ given the evidence obtained from the training sample S and prior assumptions regarding the probability

distribution of θ. Using Bayes' rule:

$$p(\theta|S) = \frac{p(S|\theta)p(\theta)}{p(S)} \Leftrightarrow p(\theta|S) = \frac{\prod_{t=1}^{n} p(y^t|x^t,\theta)p(\theta)}{p(S)}$$

The marginal probability of the sample, the training set, $p(S)$, cannot generally be computed, but we can see it as simply a normalization value that guarantees that the probability distribution $p(\theta|S)$ integrates to 1. So, again, we can ignore this probability and find the θ that maximizes the numerator in the expression above. Thus, the *maximum a posteriori*(MAP) estimate for θ is:

$$\hat{\theta}_{MAP} = \arg\max_{\theta} \prod_{t=1}^{n} p(y^t|x^t,\theta)p(\theta) \tag{14.2}$$

In practice, the difference between equation 14.1 and equation 14.2 is that the MAP estimate includes the prior probability distribution of the parameters θ. The ML approach is equivalent to MAP when the prior probability distribution of the parameters is uniform but, if we assume a non-uniform prior distribution of θ, the results are different. This not only permits the inclusion of prior assumptions regarding reasonable values of θ but also functions as a regularization term with an explicit probabilistic justification. For example, if we assume, as a prior probability distribution for θ, that all parameter values are normally distributed with a mean of 0 and a standard deviation of 1, the MAP estimate will tend to keep the θ values close to 0 and prevent them from increasing too much, as can happen with a pure maximum likelihood estimate which does not consider θ to be a random variable or have a probability distribution.

14.2 Computing Priors

The ML approach makes no assumption about the prior probability distribution of the parameters because it does not even consider the parameters to be random variables. An *uninformative prior* is a Bayesian assumption about the prior distribution of the parameters that leads to approximately the same result as the ML approach, having no significant impact on the posterior probability. In some cases, assuming a uniform prior distribution for the model parameters θ can be an *uninformative prior*. However, sometimes this is not the case. For example, in a linear regression, the slope of the line varies from zero for a horizontal line to minus or plus infinity for a vertical line. If we assume a uniform distribution for the slope we will strongly bias our prior distribution towards nearly vertical lines, since this is where the vast majority of the values will be. Thus, in many cases, assuming a uniform distribution of the parameters is not adequate. Furthermore, there are cases where we may want to use an informative prior because we do have some information about the prior probability distribution of our parameters.

This often results in prior probability distributions for which we do not have analytical solutions for means and standard deviations. Thus, in MAP parameter estimation, it is often necessary to use numerical techniques to sample these prior probability distributions and obtain the necessary statistics. This is done by Monte Carlo techniques, most commonly by Markov Chain Monte Carlo (MCMC), which computes random walks over parameter values based on the probability distribution function in order to generate the appropriate samples. VanderPlas [22] illustrates this problem and shows some solutions using Python MCMC libraries.

In practice, even when using maximum likelihood methods in machine learning, we resort to regularization, which can be seen as a way to encourage our parameters to fall within reasonable intervals even though we do not make explicit assumptions about their prior probability distributions nor consider them to be random variables. Bayesian learning provides a more rigorous alternative, but often at significant computational cost due to the need to rely on numerical sampling methods.

14.3 Decision and Costs

Aside from the question of prior assumptions, which we can deal with explicitly in a Bayesian approach, another problem that may occur when fitting parameters to create a classifier is the cost of different mistakes.

Let us consider, as a simple example, a binary classification problem in one dimension. For each class C_1 and C_2, there is a different distribution of the feature value x, with different joint probabilities $P(x, C_1)$ and $P(x, C_2)$. If we want to create a classifier that classifies an example as C_2 if $x > \hat{x}$ or C_1, then we need to find the best value for the threshold $\hat{x}$. The probability and type of each error will depend on the value of $\hat{x}$, as illustrated in Figure 14.1. One possibility would be to minimize the total probability of committing an error, as shown on the right panel of Figure 14.1.

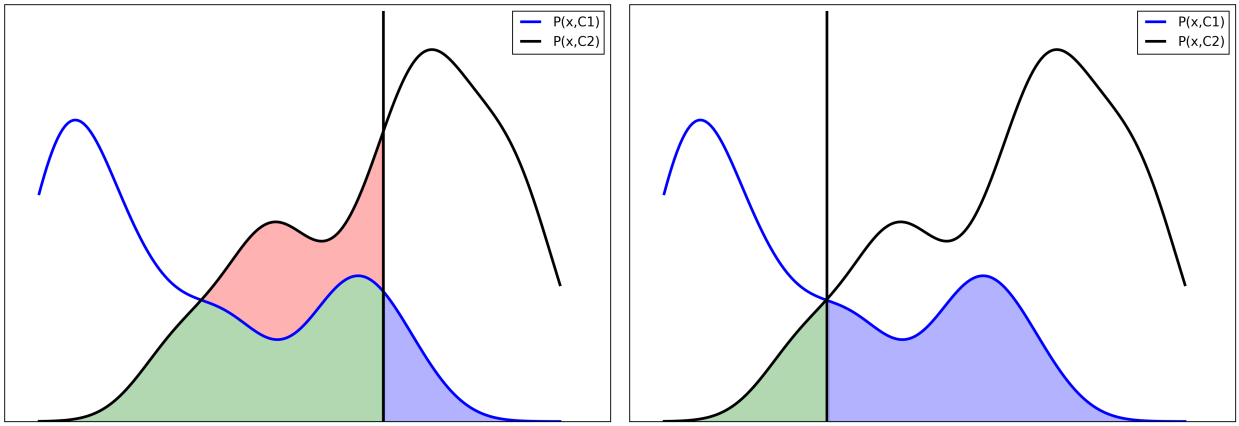

Figure 14.1: For a given value of $\hat{x}$, the threshold dividing the two classes determines the probability of each type of error. Red and green show the probability of classifying as class 1 a point of class 2 and blue the probability of classifying as class 2 a point in class 1.

However, this may not be the best option in general. Suppose class 1 corresponds to subjects with cancer and class 2 to healthy subjects. In this case, the error of misdiagnosing a healthy subject and concluding the subject has cancer is not as serious as the error of misdiagnosing a cancer patient and concluding them healthy. Let us consider the *loss matrix* shown in Table 14.1. This indicates that the cost of misdiagnosing a cancer patient is five times greater than the cost of misdiagnosing a healthy subject.

Table 14.1: Loss matrix for cancer diagnosis.

	Predict Cancer	Predict Healthy
Has Cancer	0	5
Is Healthy	1	0

Instead of trying to minimize the probability of error, we can minimize the expected loss by assigning each example to the class j that minimizes the *loss function*, which sum, for all classes k, of the joint

probability of the class multiplied by the cost of classifying the example as class j when the true class is k:

$$\arg\min_j \sum_k L_{k,j} p(C_k|x)$$

In this case, the frontier dividing the two classes may not correspond to the point of lowest error probability but, rather, to the point where the error cost is minimum, as illustrated in Figure 14.2.

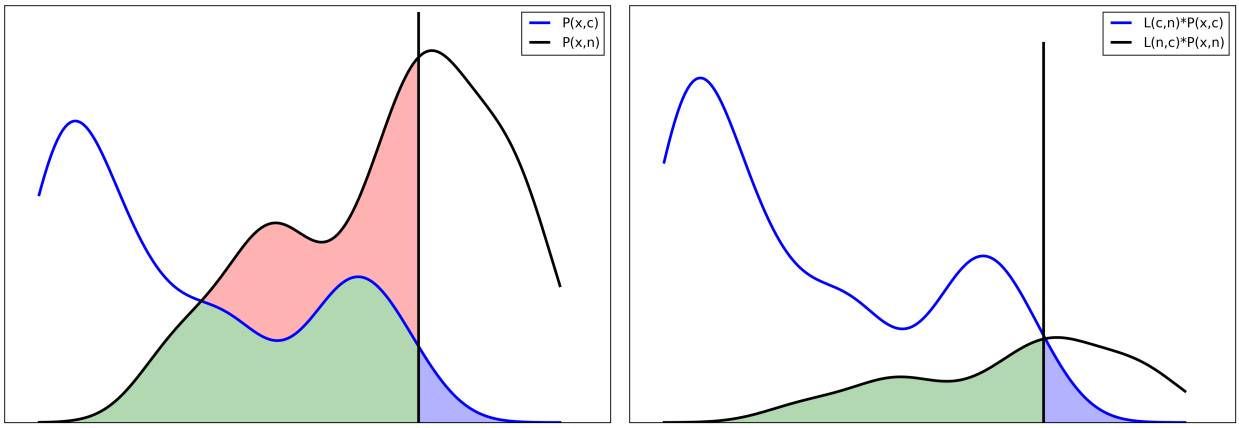

Figure 14.2: When taking into account the different costs (loss) of different errors, according to the loss matrix in Table 14.1, the frontier between the two classes gets shifter by the cost values. In this simple example, this can be understood as finding the minimum point of the probability curves multiplied by the misclassification costs (right panel).

Another problem with classification and learning is how to deal with the different levels of certainty in deciding which class or value to predict. In some cases, the predicted probability of an example belonging to one class may be only marginally larger than the probability of belonging to another class, which makes the prediction much less reliable than it would be if one probability was large and the remaining small. Figure 14.3 illustrates this problem. For low values of x, the class represented by the blue line has a much larger probability than the class represented by the black line, with the converse for high values of x. But in the mid range, the probabilities of both classes approach and a decision there is less reliable. One way to avoid this problem is to abstain from offering a classification when the probabilities for all classes, k, are below some threshold:

$$p(C_k|x) \leq \phi \qquad \forall k$$

. In the figure, this would be the 0.7 threshold. For these cases, no classification is given. This is called the *reject option*.

14.4 Further Reading

1. Alpaydin [2], Chapter 3 up to sections 3.5
2. Bishop [4], Section 1.5
3. VanderPlas, Jake, Frequentism and bayesianism: a python-driven primer, [22]

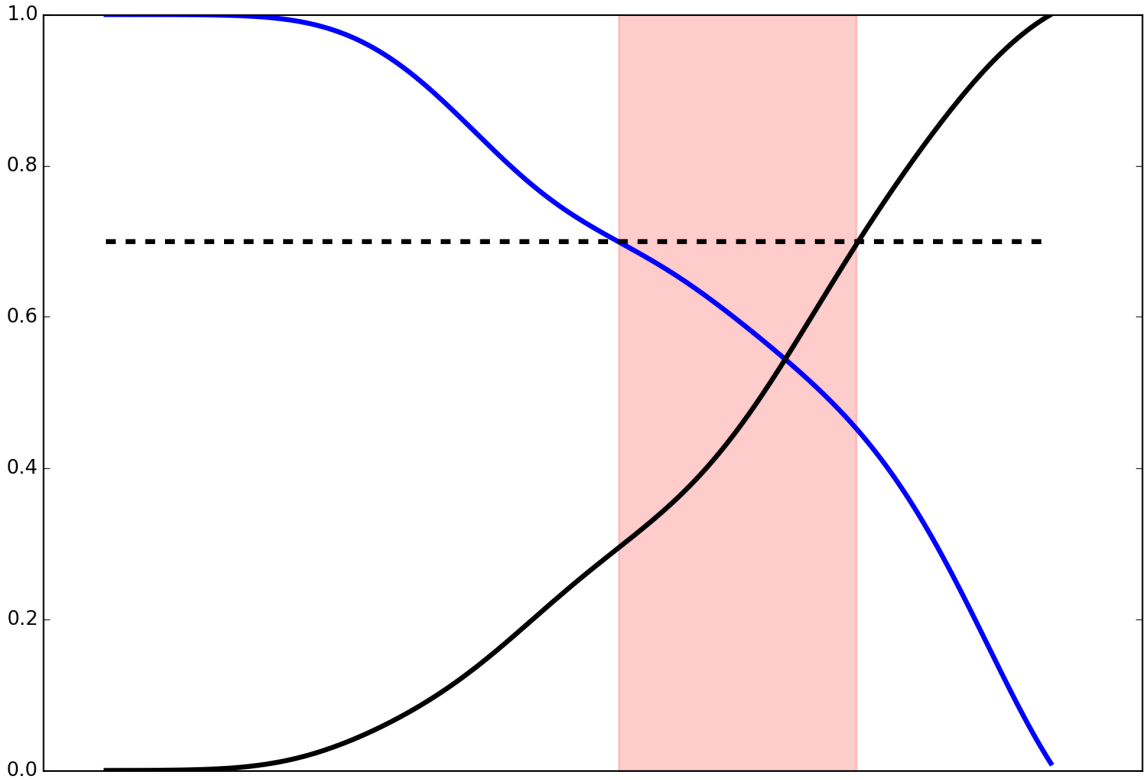

Figure 14.3: The probability of an example being in each class (blue and black) as a function of the feature value x. The region marked in red is the reject region, in which the classifier will not propose any classification because the probability for all classes is below the predefined threshold (0.7 in this case).

Part III

Unsupervised Learning

Chapter 15

Introduction to Unsupervised Learning

Introduction to unsupervised learning. Data visualization and feature selection.

15.1 Unsupervised Learning

Unsupervised learning, simply put, is the process of adjusting some model to the structure of the data without relying on an error measure evaluated with reference to known labels. This does not mean the data cannot have known labels. It is quite possible, and often useful, to use unsupervised learning with labelled data. But, unlike supervised learning, the goal of unsupervised learning is simply to describe aspects of the data — how it is distributed, relations between features and so forth — instead of trying to predict some value that is known for the training set and which can be used to supervise learning. In other words, the goal of unsupervised learning is to transform the data into some representation that makes it easier to understand it or use it for some other purpose, like supervised learning. Figure 15.1 illustrates this process.

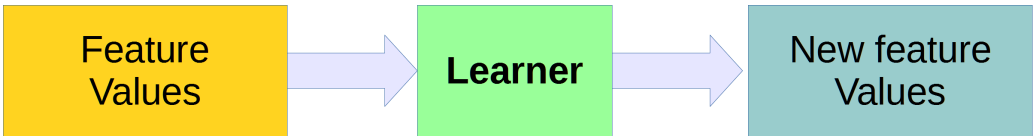

Figure 15.1: Diagram representing unsupervised learning.

Although, strictly speaking, data visualization by itself may not include unsupervised learning, as it may not involve learning at all, it is a good starting point to understand the purpose of unsupervised learning. So we shall begin with the problem of visualizing data with more than two dimensions.

15.2 Visualizing Data

We shall consider, as an example, the Iris dataset, first introduced by Fisher in 1936 [1]. Figure 15.2 shows examples of the three classes of flowers. Each flower is described by four features, the length and width of sepals and petals.

[1]The dataset can be downloaded from the MIST repository: https://archive.ics.uci.edu/ml/datasets/Iris

Figure 15.2: The Iris data set contains values for sepal and petal lengths and widths for flowers of three Iris species. Images CC BY-SA. Authors Setosa: Szczecinkowaty; Versicolor: Gordon, Robertson; Virginica: Mayfield.

The problem is how to visualize this four-dimensional data, since we can only understand three dimensions and, for practical purposes, two-dimensional representations are preferable. For this purpose, we will use the Python Data Analysis (Pandas) library, since it includes many convenient features for image visualization [2].

We begin by reading the `.csv` data file using the `read_csv` function from the Pandas library and then plot the histograms of the features in a single figure. This is the file format:

```
SepalLength,SepalWidth,PetalLength,PetalWidth,Name
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
...
5.1,2.5,3.0,1.1,Iris-versicolor
5.7,2.8,4.1,1.3,Iris-versicolor
...
6.2,3.4,5.4,2.3,Iris-virginica
5.9,3.0,5.1,1.8,Iris-virginica
```

This is the code for creating the histogram.

```
from pandas import read_csv
import matplotlib.pyplot as plt

data = read_csv('iris.data')
plt.figure(figsize=(12,8))
data.plot(kind='hist', bins=15, alpha=0.5)
plt.savefig('L15-stackedhist.png', dpi=200,bbox_inches='tight')
plt.close()
```

Alternatively, instead of using the `plot` method of the class returned by the `read_csv` function to plot the histogram of all features in the same chart, we can plot the different histograms separately by using the `hist` method instead:

```
...
data.hist(color='k', alpha=0.5, bins=15)
...
```

[2] More information available on http://pandas.pydata.org/pandas-docs/stable/visualization.html

The resulting images can be seen in Figure 15.3

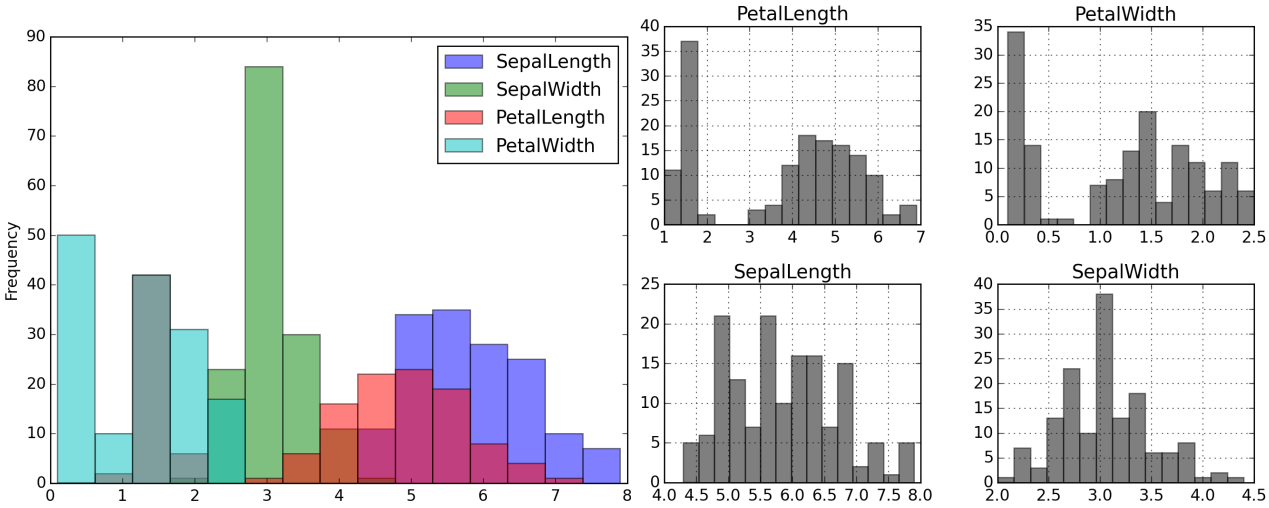

Figure 15.3: Examples of histograms for the Iris dataset. On the left, histograms of the four features in the same chart. On the right, separated in four charts.

Another way of visualizing the distribution of each feature is using a *box plot*. In this type of plot, the box represents the range between the first and third quantiles (25% and 75%), a line represents the median value and the "whiskers" are placed away from the first and third quantile values at a distance equal to the difference between these two quantile values multiplied by a constant parameter, often 1.5. We can do this with the Pandas library by using the `kind='box'` argument on the plot method. The result is shown in Figure 15.4.

```
5 ...
6 data.plot(kind='box')
7 ...
```

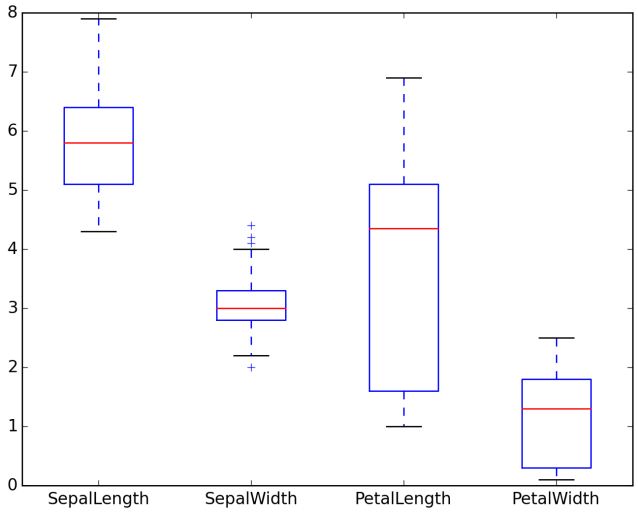

Figure 15.4: Box plot of the four features for the Iris dataset.

While histograms and box plots are useful to represent the distribution of each isolated feature, they give us no idea about how features correlate. One alternative plot to visualize correlations between pairs of features is the *scatter matrix* plot. This is a two-dimensional array of plots representing the

histogram or kernel density estimation plot of each feature in the diagonal and scatter plots of each feature as a function of another in the remaining plots. Figure 15.5 illustrates these plots for the four features in the Iris dataset. We can do this easily with the Pandas library sith the `scatter_matrix` function:

```
from pandas import read_csv
import matplotlib.pyplot as plt
from pandas.tools.plotting import scatter_matrix
data = read_csv('iris.data')
scatter_matrix(data.ix[:,[0,1,2,3]], alpha=0.5,figsize=(15,10), diagonal='kde')
plt.savefig('L15-scatter.png', dpi=200,bbox_inches='tight')
plt.close()
```

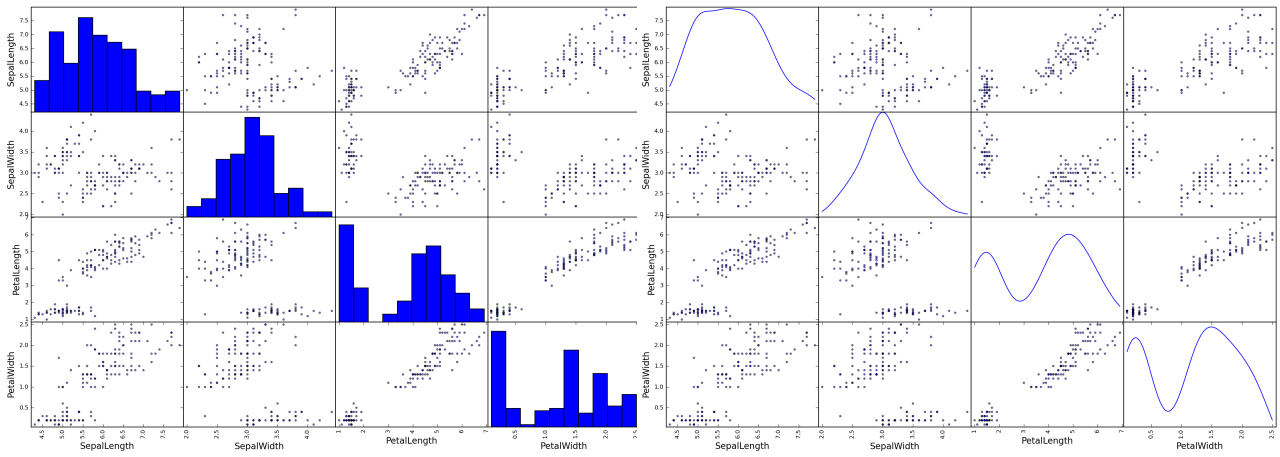

Figure 15.5: Scatter matrix plot examples, with histograms and Kernel Density Estimation in the diagonals.

Another useful method for visualizing multidimensional data is the *parallel coordinates plot*, in which features are represented as a set of parallel axes and each data point is represented as a set of line segments intersecting the feature axes at the corresponding values for each feature. The code below shows how we can do this to plot all the Iris data in single category by adding a new column, titled `'All'`, in which all points have the value `'Iris'`. Then we do the `parallel_coordinates` plot using this new column as the category field.

```
from pandas import read_csv
from pandas.tools.plotting import parallel_coordinates
import matplotlib.pyplot as plt
data = read_csv('iris.data')
all_data=data.ix[:,[0,1,2,3]]
all_data['All']='Iris'
plt.figure(figsize=(12,8))
parallel_coordinates(all_data,'All',color='b')
plt.savefig('L15-parallel-all.png', dpi=200,bbox_inches='tight')
plt.close()
```

Alternatively, we can plot the different categories in different colors by using the `'Name'` column for the category and giving three color labels in the `color` argument of the `parallel_coordinates` function. The result is shown in Figure 15.6.

```
...
parallel_coordinates(data, 'Name', color=('r','g','b'))
...
```

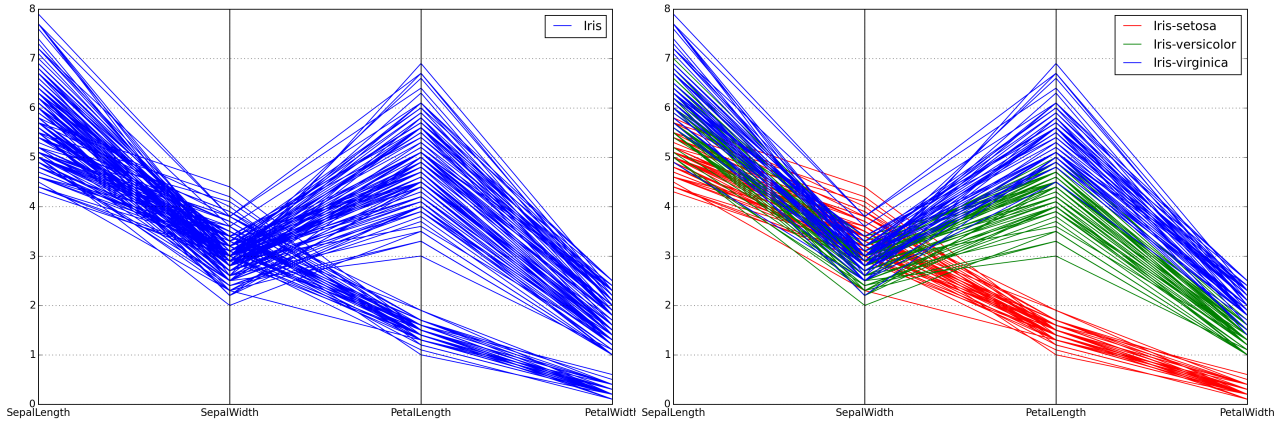

Figure 15.6: Parallel coordinates plot. Each point is represented as a line crossing the feature axes at the respective feature values. In the right panel, the lines are coloured by class.

A similar method is to use *Andrew's curves* [3]. In this plotting method, each data point is converted into a line resulting from the sum of trigonometric terms of different frequencies. Given a data point $\vec{x} = \{x_1, x_2, x_3, ...\}$, the resulting line is:

$$f_{\vec{x}}(t) = \frac{x_1}{\sqrt{2}} + x_2 \sin(t) + x_3 \cos(t) + x_4 \sin(2t) + x_5 \cos(2t) + x_6 \sin(3t) + x_7 \cos(3t)...$$

The result is that different features contribute to different frequencies on the curve, and points with similar features result in similar curves. With the Pandas library, these curves can be plotted using the `andrews_curves` function:

```
from pandas import read_csv
import matplotlib.pyplot as plt
from pandas.tools.plotting import andrews_curves
data = read_csv('iris.data')
plt.figure(figsize=(12,8))
andrews_curves(data, 'Name', color=('r','g','b'))
plt.savefig('L15-andrews.png', dpi=200,bbox_inches='tight')
plt.close()
```

The *Radial Visualization* (RADVIZ) method [13] represents each multidimensional data point as a point in two dimensions, but places the points by spreading the feature axes radially and using the value of each feature to "pull" the point in the corresponding direction. The position of the point results from the outcome of all these "forces" pulling it in different directions. This way, points that have a balanced distribution of values across the features tend to be in the middle of the plot, whereas points that favour some feature over the others are pulled by that feature's axis. This can be done with the `radviz` function of the Pandas library. The Andrew's curves and RADVIZ plots are shown in Figure 15.7.

```
from pandas import read_csv
import matplotlib.pyplot as plt
```

```
3 from pandas.tools.plotting import radviz
4 data = read_csv('iris.data')
5 plt.figure(figsize=(12,8))
6 radviz(data, 'Name', color=('r','g','b'))
7 plt.savefig('L15-radviz.png', dpi=200,bbox_inches='tight')
8 plt.close()
```

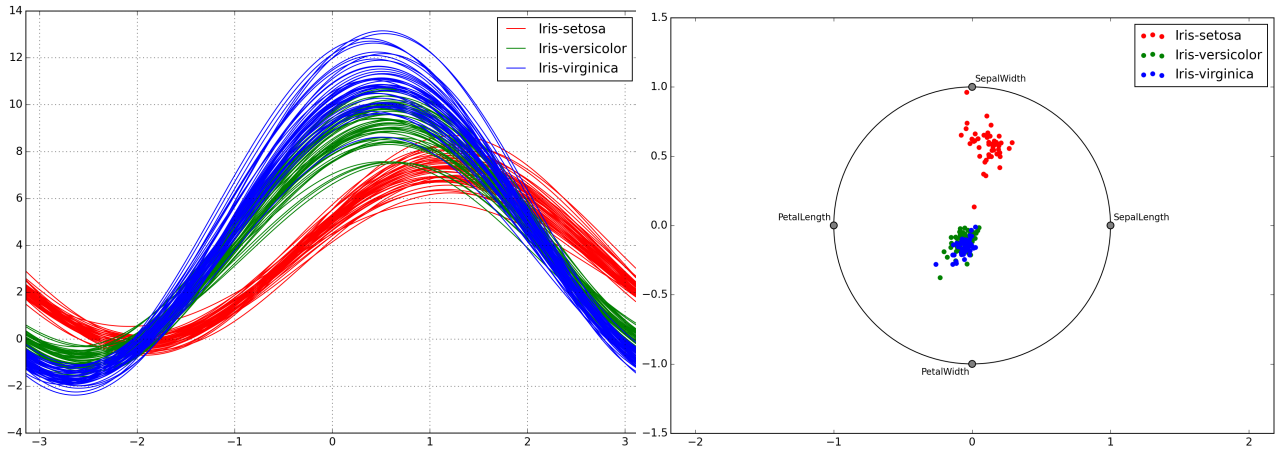

Figure 15.7: Andrews curves and RADVIZ plot. Data coloured by class.

15.3 Feature Selection

In general, not all features are equally useful for learning and it may be beneficial to reduce the dimensionality of the training set, both for supervised and unsupervised learning. There may be several reasons for this. There may be too many features for the available data, leading to overfitting; some features may be too noisy or uninformative; some features may be costly to measure and so forth. One way of reducing the number of features is to discard all but the best. This is *feature selection* and can be done by examining and discarding features before the learning process, or according to the performance of the hypotheses learned or even as an integral part of the learning process. Discarding features before beginning to train the learner is called filtering, an can be either *univariate filtering* if the features are discarded by examining each feature individually or *multivariate filtering* if features are examined jointly with other features.

Univariate filtering is easier to understand when we are dealing with labelled data and want to prepare the data for supervised learning. In this case, we can select features by comparing each feature with the data labels. One criterion for selecting features in this case can be the statistical independence of each feature and the class, since features that are statistically independent from the class are not useful for predicting the class. Statistical independence can by assessed by the χ^2 (chi-squared) test, a generic test that gives us the probability of obtaining some sample when drawing at random from some distribution. If O_n are the observed frequencies and E_n the expected frequencies, the chi-squared value is:

$$\chi^2 = \sum_{i=1}^{N} \frac{(O_i - E_i)^2}{E_i}$$

If we have a feature with K categorical values and a classification problem with C classes, we can compute the observed number of cases where the feature has a value k in points with class c, O_{kc}, and

the expected number E_{kc} assuming the feature and class are independent, which is obtained from the fraction of value k and class c. In this case, the chi-squared value (for $(K-1)(C-1)$ degrees of freedom) is:

$$\chi^2 = \sum_{k=1}^{K}\sum_{c=1}^{C}\frac{(O_{kc}-E_{kc})^2}{E_{kc}}$$

Using the chi-squared test we can eliminate those features that, having a low χ^2 value, are closer to being statistically independent of the class.

Another statistical test for labelled data is the *Analysis of Variance* (ANOVA) F-test, which compares the variance between groups with the variance within the groups. Again, this proportion has a known probability distribution under the assumption that the variables are independent, and thus can be used to find the likelihood of that assumption. If the F-test value is low, and thus the likelihood of independence is high, we can reject the feature as uninformative. The code below shows how to use the ANOVA F-test with Scikit-Learn library, on the Iris dataset:

```
1 from sklearn.feature_selection import f_classif
2 from sklearn import datasets
3
4 iris = datasets.load_iris()
5 X = iris.data
6 y = iris.target
7 f,prob = f_classif(X,y)
8 print f
9 print prob
```

The F-test values and respective probabilities indicate which features deviate the most from being independent of the class (those with the smallest probability values). Figure 15.8 shows the scatter plot of the two best features from the Iris dataset, according to the F-test, which are the two features with the lowest F-test probabilities.

These methods rely on labelled data, and determine the relevance of each feature for predicting the labels. A feature is *relevant* if it correlates to the labels, and irrelevant if it is independent of the labels, in which case we discard it. But we can also filter features according to their correlation to other features, because a feature is *redundant* if it correlates to another features. This requires filtering features by comparing them to each other, which is called *multivariate fitering*. In this approach, if several features are strongly correlated one to the others, we can discard all but one of the set, since the information given by that one is nearly the same as that given by all other correlated features. Since this can be applied both to labelled and unlabelled data sets, it can be more useful for unsupervised learning.

Instead of filtering features prior to training, we can also use the performance of the trained hypotheses to evaluate the adequacy of the set of features used. These are called *wrapper methods* for feature selection, and consist of a scoring algorithm, typically the machine learning algorithm we wish to use, and a search algorithm that runs the learning algorithm on subsets of all features to find the best subset. For this we can use a *deterministic wrapper* that iterates through all possible subsets. With *sequential forward selection* we start with the empty set and, at each iteration, loop through all remaining features to find the best one to add to that set, according to the performance of our learning algorithm. This is repeated until we reach the desired number of features or performance level. With *sequential backward elimination* we do the search in the opposite direction, starting with all features

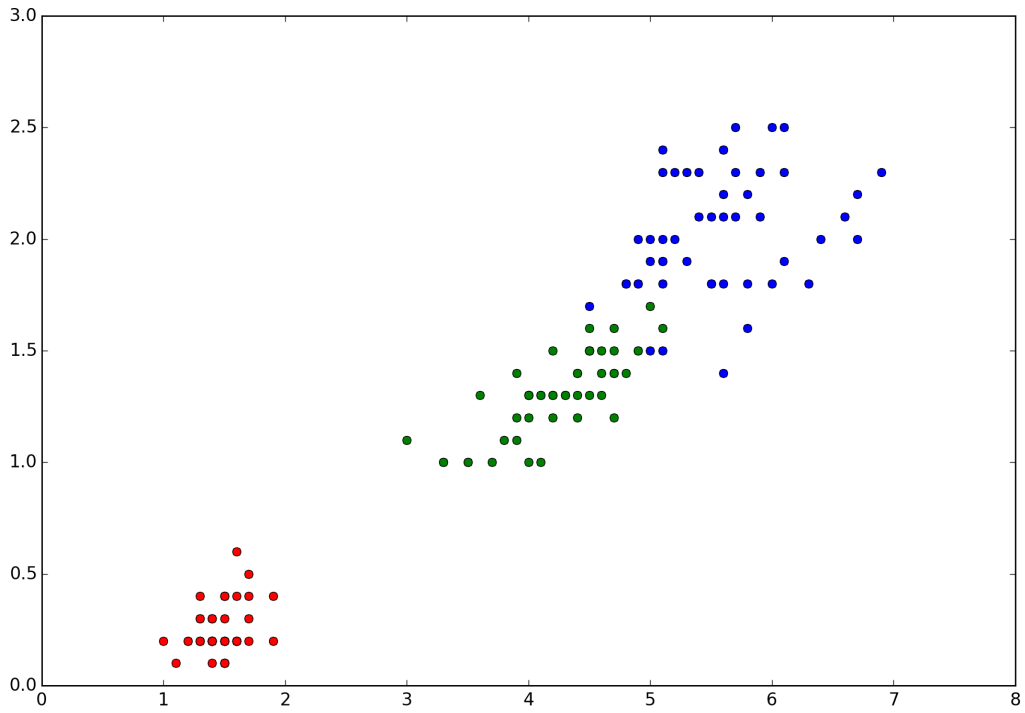

Figure 15.8: Scatter plot of the two best features (petal length and petal width) according to the ANOVA F-Test.

and removing, at each iteration, we eliminate one feature so that the performance of the classifier is maximized.

Alternatively, we can also use a *non-deterministic wrapper* that searchs the subsets of features with non-deterministic algorithms such as genetic algorithms or simulated annealing. The procedure is the same, trying to maximize the performance with a limited number of features, but without the greedy search of the deterministic wrapper methods.

Finally, some learning algorithms incorporate feature selection. This is called *embedded feature selection*. Decision trees with a limited depth are an example of this kind of algorithm, since the best features are used earlier in the tree and, by limiting the tree depth, less useful features end up being ignored. Naïve Bayes with weighted features is another example. For example, features may be weighted according to how much the conditional distribution of the feature values given a class differs from the prior probability of the class, which indicates more relevant features [14]. Embedded feature selection can also be done through regularization. For example, using L1 regularization, which penalizes the sum of the absolute values of the parameters. This forces some parameters to be 0, effectively ignoring the corresponding features. Logistic Regression in Scikit-Learn can be done with L1 regularization.

15.4 Further Reading

1. Pandas library visualization tutorial: `http://pandas.pydata.org/pandas-docs/stable/visualization.html`

2. Scikit-Learn feature selection tutorial: `http://scikit-learn.org/stable/modules/feature_selection.html`

3. Alpaydin [2], Sections 6.2. and 6.9

4. A review of feature selection techniques in bioinformatics [19]

Chapter 16

Feature Extraction

Dimensionality reduction: feature extraction with PCA; self-organizing maps.

16.1 Dimensionality Reduction

In Chapter 15 we saw how to reduce the dimensionality of a data set by selecting only a subset of features, whether by filtering, using a wrapper to evaluate the performance of the learner for each subset of features or using learning algorithms that embed feature selection. In this chapter, we will see a different approach, *feature extraction*, which consists of computing new features using a function of the original features in the dataset. This approach is very useful in many cases, such as image processing, text mining or voice recognition. The main idea is to transform the original data into a more useful data set.

There are many domain-specific algorithms for feature extraction. Identifying regions of interest in an image requires different methods from extracting specific frequencies from a sound file, for example. But in this chapter we will focus on some generic approaches that do not depend on the type of problem. One widely used, and useful, approach is *Principal Component Analysis* (PCA).

16.2 Principal Component Analysis

Formally, PCA is a procedure for finding a transformation of a data set into an orthogonal set of coordinates chosen so that the values along each new coordinate are linearly uncorrelated. Another way of imagining PCA, is that we are going to choose the direction along which the data points have the greatest variance — that is, are more "spread out" — and then project the data in this direction, the *principal component*. Then we iteratively choose a new direction, orthogonal to all previous ones, using the same criterion of maximum variance.

Figure 16.1 illustrates this process. On the left panel, we see a set of points in three dimensions, and can note that the distributions over the different coordinates are not uncorrelated, since the point cloud is spread along a diagonal. If we compute the vector along this diagonal, one of the three vectors represented in red, and project the data in that direction, we can then find the next *principal component* by doing the same computation on the projected data. We can imagine repeating this proces until there is only one orthogonal direction left, giving the third vector in this case, since we started from three dimensions. The panel on the right shows the result of projecting the three-dimensional data into

the first two *principal components*. Note that, after this transformation, the coordinates are no longer linearly correlated, with the points no longer spread along a diagonal.

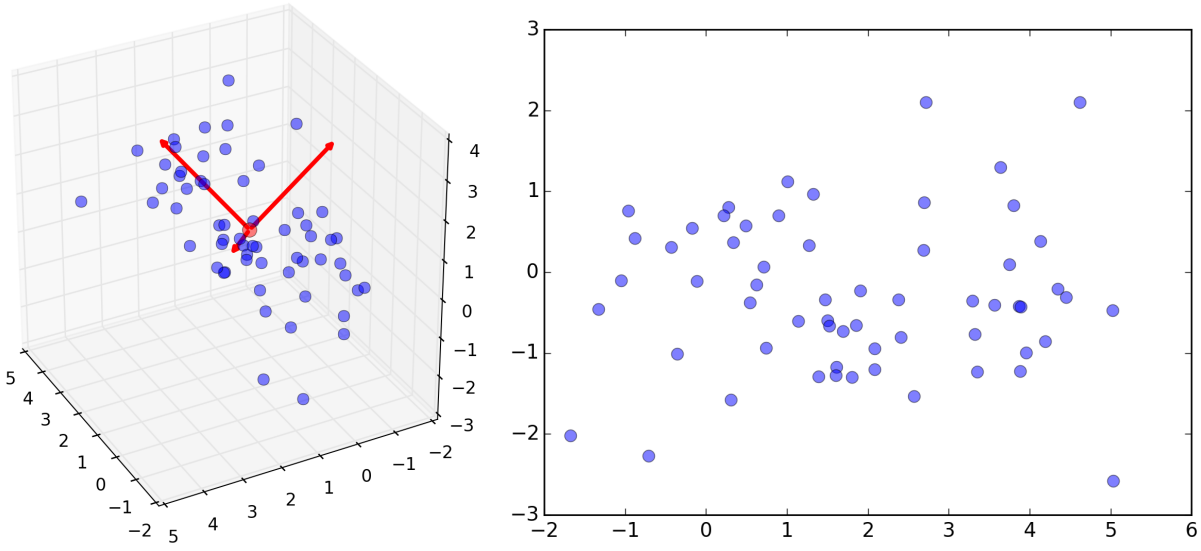

Figure 16.1: The left panel shows a three-dimensional data set with correlated coordinates, and corresponding three principal components. The right panel shows the projection of the original point into the first two principal components.

In practice, we do not compute the principal components in this iterative manner. This is just to make it easier to imagine the process. The way PCA is done is by computing the *eigenvectors* of the *covariance matrix* or, more precisely, of the *scatter matrix*[1]. The scatter matrix S can be computed by adding the matrices obtained by the outer products of all data vectors with themselves, after subtracting the mean vector:

$$m = \frac{1}{n}\sum_{k=1}^{n} x_k \qquad S = \sum_{k=1}^{n}(x_k - m)(x_k - m)^T$$

Using the Numpy library, we can compute the scatter matrix by computing the mean vector and then the outer products of the data points minus the mean vector. Note that, with the Numpy library, the mean vector cam be computed in a single instruction. This implementation is only to make it clearer how the vector is computed.

```
1  import numpy as np
2  mean_x = np.mean(data[0,:])
3  mean_y = np.mean(data[1,:])
4  mean_z = np.mean(data[2,:])
5  mean_v = np.array([[mean_x],[mean_y],[mean_z]])
6  scatter = np.zeros((3,3))
7  for i in range(data.shape[1]):
8      scatter += (data[:,i].reshape(3,1) - mean_v).dot((data[:,i].reshape(3,1) - mean_v).T)
9
10 print mean_v
11 [[ 1.07726488]
```

[1]The scatter matrix divided by the number of samples is the maximum likelihood estimator of the covariance matrix but, for our purposes, this scaling factor is not important, so we can use the scatter matrix directly. This explanation is based on the PCA demo authored by Sebastian Raschka, available at http://sebastianraschka.com/Articles/2014_pca_step_by_step.html

```
 [ 1.11609716]
 [ 1.03600411]]
print scatter
 [[ 110.10604771   39.91266264   52.3183266 ]
  [  39.91266264   80.68947748   34.48293948]
  [  52.3183266    34.48293948   97.58136923]]
```

Once we have the scatter matrix, we can compute the *eigenvectors* and corresponding *eigenvalues*. The eigenvectors of a matrix are those vectors which, after multiplication by the matrix, retain the same direction, changing only by a scalar factor. Thus, if v is an eigenvector of matrix A,

$$Av = \lambda v$$

The scaling factor λ is the corresponding *eigenvalue*, which can be used to sort the *eigenvectors* in order to give us the principal components in order of importance. The details of this computation fall outside the scope of this course, but we can use the `eig` function from the `linalg` module in the Numpy library. This function returns a vector with the eigenvalues and a matrix with the corresponding normalized eigenvectors, in columns (the first column of the matrix is the eigenvector corresponding to the first eigenvalue, and so on):

```
eig_vals, eig_vecs = np.linalg.eig(scatter)
print eig_vals
[ 183.57291365   51.00423734   53.79974343]
print eig_vecs
[[ 0.66718409  0.72273622  0.18032676]
 [ 0.45619248 -0.20507368 -0.8659291 ]
 [ 0.58885805 -0.65999783  0.46652873]]
```

The two largest eigenvalues are, in order, the first and the third. This means that these are the first two principal components of our data set, and the two best directions do choose to project the three-dimensional data into two dimensions, as shown in Figure 16.1. To do this, we combine these two vectors into a transformation matrix, then transform the data and plot it.

```
transf = np.vstack((eig_vecs[:,0],eig_vecs[:,2]))
t_data = transf.dot(data.T)
fig = plt.figure(figsize=(7,7))
plt.plot(t_data[0,:], t_data[1,:], 'o', markersize=7, color='blue', alpha=0.5)
plt.gca().set_aspect('equal', adjustable='box')
plt.savefig('L16-transf.png',dpi=200,bbox_inches='tight')
plt.close()
```

By plotting the first principal component in the x axis we get most of the variance in this axis, with the values ranging from -2 to 6. The second principal component, in the y axis, corresponds to the direction, orthogonal to the first, that has the largest of the remaining variance. In this case, the range is now only from -3 to 3. It is also worth noting that the projected points are no longer in a diagonal distribution, as the new coordinates now are linearly uncorrelated due to the transformation using the principal components.

The `decomposition` module of the Scikit-Learn library offers classes `PCA` and `RandomizedPCA` for principal component analysis. The `RandomizedPCA` is suitable for large datasets, using random samples of the data for the PCA instead of the complete dataset.

16.3 Self Organizing Maps

Another way of projecting a high dimension data set into a smaller set of dimensions is to use a *Self Organizing Map* (SOM). We can imagine the SOM as an artificial neural network whose neurons are arranged in a two-dimensional matrix, with each neuron in the SOM having a set of coefficients of the same dimension as the data set. This gives us two distance measures: we can measure the distance from the coefficients vector of any neuron to any point in the data set, and we can measure the distance within the neuron matrix from any neuron to its neighbours.

The SOM is trained by first assigning random values to the coefficients of the neurons. Then, iteratively, we start by finding the neuron closest to a data point, called the *Best Matching Unit* (BMU), and shifting the coefficients vector of the BMU neuron closer to the data point. Neurons that are close to the BMU in the SOM matrix are also moved in the same direction, though by a smaller amount decreasing with the distance to the BMU in the SOM matrix. The magnitude of these changes is a function of a learning coefficient that decreases monotonically during training. Figure 16.2

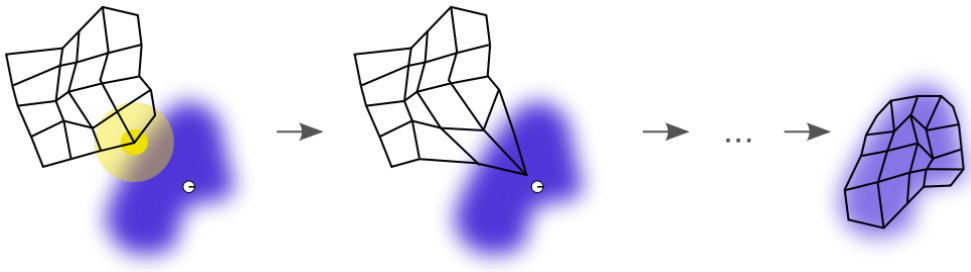

Figure 16.2: Training the SOM. As the coefficient vector of each neuron is changed, it "pulls" on the vectors of neighbouring neurons, making the neuron matrix adjust to the data set in the space of the data points. Image source: Wikipedia

To illustrate the use of a SOM, we will project the three-dimensional colour space into a two-dimensional matrix. Each colour is defined by a vector of 3 values, for the red, green and blue components. We will use the minisom module[2] to train a SOM of 20 by 30 neurons, for a total of 600 neurons[3]. We start by creating a labelled set of colors,

```
colors = np.array(
         [[0., 0., 0.],
          [0., 0., 1.],
          ...
          [.5, .5, .5],
          [.66, .66, .66]])
color_names = \
        ['black', 'blue', 'darkblue', 'skyblue',
         'greyblue', 'lilac', 'green', 'red',
         'cyan', 'violet', 'yellow', 'white',
         'darkgrey', 'mediumgrey', 'lightgrey']
```

The `MiniSom` class is initialized by providing the dimensions of the SOM. In order, the number of neurons in the x and y dimensions and the dimension of the input space. The `learning_rate` is the

[2] Available at https://github.com/JustGlowing/minisom

[3] This example is based on a SOM demo at the Multivariate Pattern Analysis in Python site: http://www.pymvpa.org/examples/som.html

multiplier for the adjustment in the neuron coefficients and `sigma` is a parameter defining the neighbourhood function on the SOM matrix. The methods `random_weights_init` and `train_batch` serve, respectively, to initialize the coefficients and train the SOM. The initialization consists of assigning random points from the training set to the SOM neurons as coefficients.

```
from minisom import MiniSom
import matplotlib.pyplot as plt
import numpy as np

plt.figure(1, figsize=(7.5, 5), frameon=False)
som = MiniSom(20, 30, 3, learning_rate=0.5, sigma = 2)
som.random_weights_init(colors)
som.train_batch(colors,10000)
```

To view the result, we can draw the matrix using the colours corresponding to the three coefficients of each SOM neuron, each coefficient corresponding to a colour channel. We can also draw the colour labels on the SOM matrix by placing them at the position of the SOM neuron whose coefficients are closer to the colour values. To do this, we use the `winner` method of the SOM object to obtain the coordinates, in the SOM matrix, of the Best Matching Unit for the colour vector. The code below details this process and Figure 16.3 shows the resulting image.

```
for ix in range(len(colors)):
    winner = som.winner(colors[ix])
    plt.text(winner[1], winner[0], color_names[ix], ha='center', va='center',
             bbox=dict(facecolor='white', alpha=0.5, lw=0))
plt.imshow(som.weights, origin='lower')
plt.savefig('L6-colors.png',dpi=300)
plt.close()
```

16.4 An example of feature extraction

To illustrate the process of feature extraction and data projection with a SOM, we'll examine data from the Gapminder site[4]. We have data on a set of indicators: *per capita* GDP, life expectancy, infant mortality and unemployment. Each indicator is available in an Excel spreadsheet file with one year in each column and one country in each row. Figure 16.4 illustrates the structure of these files.

The problem here is that the data is not uniform in quality. For each country and indicator there may be data for some years and not others, so there are different numbers of data points for different countries, as illustrated in Figure 16.5. This makes it hard to organize the information. So the first step will be to extract from these heterogeneous sets of data a set of features with a fixed dimension for all countries. We can do this by fitting each curve with a third degree polynomial. This will allow us to represent each country as a set of 16 features, with four features for the curve of each four indicators. Figure 16.6 shows examples of polynomial curves obtained from the standardized indicator values, with years and indicator values rescaled to a range of $[0, 1]$.

With this dataset with 16 dimensions, with a 16 dimensional vector describing each country, we can train a SOM in order to project the countries into a two-dimensional image according to their similarity in the pattern of the four indicators. We start by normalizing these data by subtracting the mean value

[4] http://www.gapminder.org

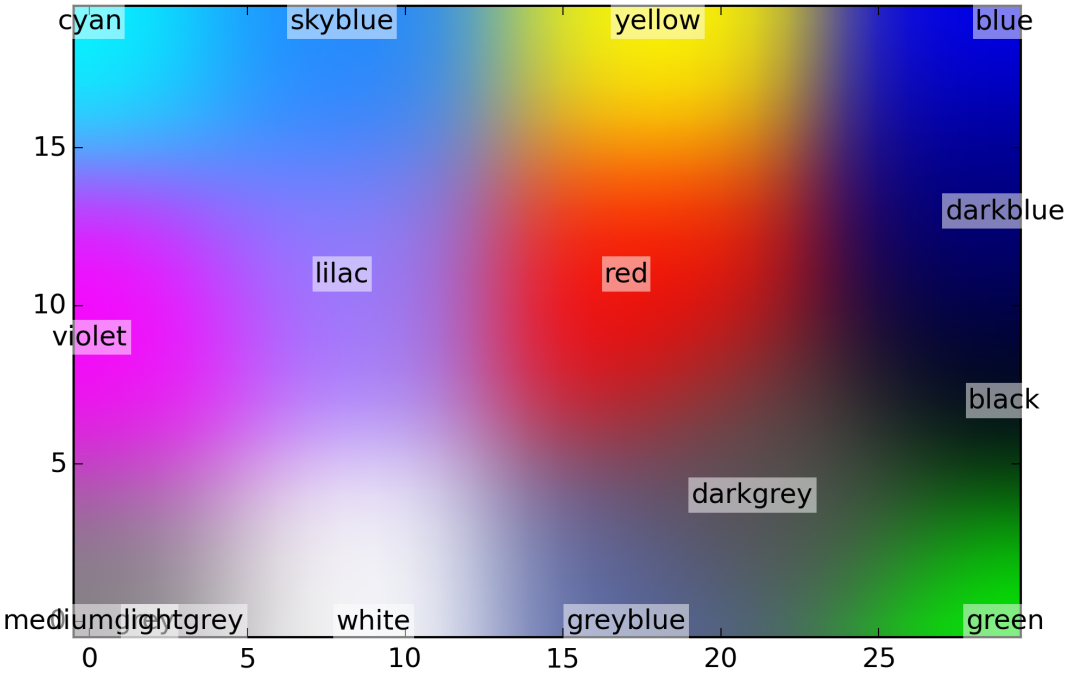

Figure 16.3: Result of training the SOM with the set of colours and labelling the colours at the respective SOM neurons.

BK25 28,799

	A	AR	AS	AT	AU	AV	AW	AX	AY
1	Life expectancy with projections	1807	1808	1809	1810	1811	1812	1813	1814
2	Afghanistan	28,139273	28,129027	28,11878	28,108533	28,098287	28,08804	28,077793	28,067547
3	Albania	35,4	35,4	35,4	35,4	35,4	35,4	35,4	35,4
4	Algeria	28,8224	28,8224	28,8224	28,8224	28,8224	28,8224	28,8224	28,8224
5	American Samoa								
6	Andorra								
7	Angola	26,98	26,98	26,98	26,98	26,98	26,98	26,98	26,98
8	Anguilla								
9	Antigua and Barbuda	33,536	33,536	33,536	33,536	33,536	33,536	33,536	33,536
10	Argentina	33,2	33,2	33,2	33,2	33,2	33,2	33,2	33,2
11	Armenia	33,995	33,995	33,995	33,995	33,995	33,995	33,995	33,995
12	Aruba	34,419	34,419	34,419	34,419	34,419	34,419	34,419	34,419
13	Australia	34,05	34,05	34,05	34,05	34,05	34,05	34,05	34,05
14	Austria	34,4	34,4	34,4	34,4	34,4	34,4	34,4	34,4
15	Azerbaijan	29,165	29,165	29,165	29,165	29,165	29,165	29,165	29,165

Figure 16.4: Data available for each indicator.

of each feature and dividing by the standard deviation. This is necessary because the coefficients of the polynomials can span a wide range of values.

```
descs = np.zeros((len(countries),len(data_names)*(degree+1)))
features = len(data_names)*(degree+1)
for ix in range(len(countries)):
    c = countries[ix]
    c_desc = c.descriptors.reshape((1,features))
    descs[ix,:] = c_desc
descs = (descs-np.average(descs,axis=0))/np.std(descs,axis=0)
```

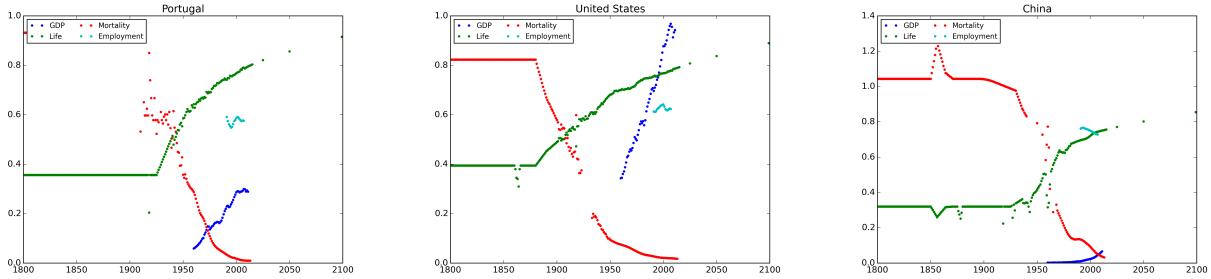

Figure 16.5: Examples of the data points available for the four indicators in three different countries

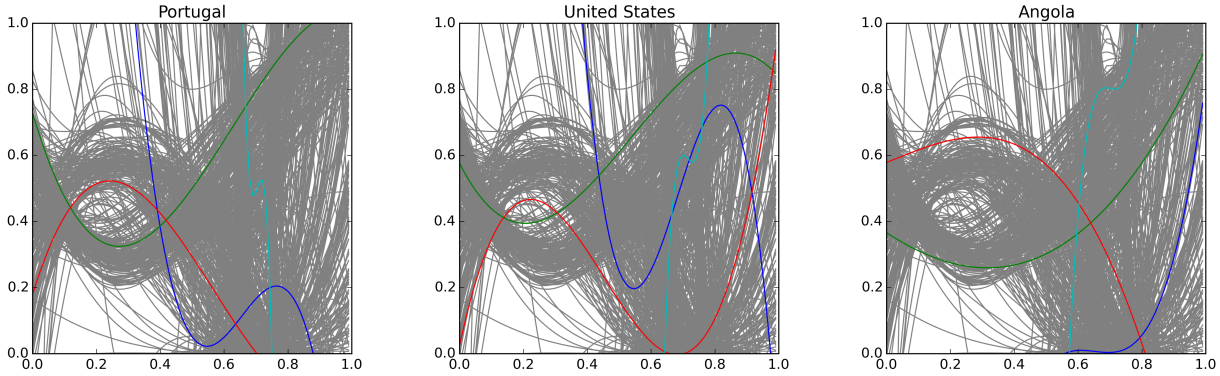

Figure 16.6: Polynomial curves adjusted to the indicator data points

Then we train the SOM and read a list with the set of countries to label on the neuron matrix.

```
1 som = MiniSom(30, 45, features, learning_rate=0.5, sigma = 2)
2 som.random_weights_init(descs)
3 som.train_batch(descs,10000)
4 to_plot = open('countries_to_plot.txt').readlines()
5 for ix in range(len(to_plot)):
6     to_plot[ix]=to_plot[ix].strip()
```

Finally, we can represent the SOM colouring each neuron on the matrix in a lighter colour the larger its average distance to its neighbours. Figure 16.7 shows the result, indicating the position in the SOM of the neurons closest to the selected countries.

```
1 plt.figure(1, figsize=(7.5, 5), frameon=False)
2 plt.bone()
3 plt.pcolor(som.distance_map()) # average dist. to neighs.
4 for ix in range(len(descs)):
5     if countries[ix].name in to_plot:
6         winner = som.winner(descs[ix])
7         plt.text(winner[1], winner[0], countries[ix].name,
8                  ha='center', va='center',color='lime')
9 plt.savefig('L6-countries_som.png',dpi=300)
10 plt.close()
```

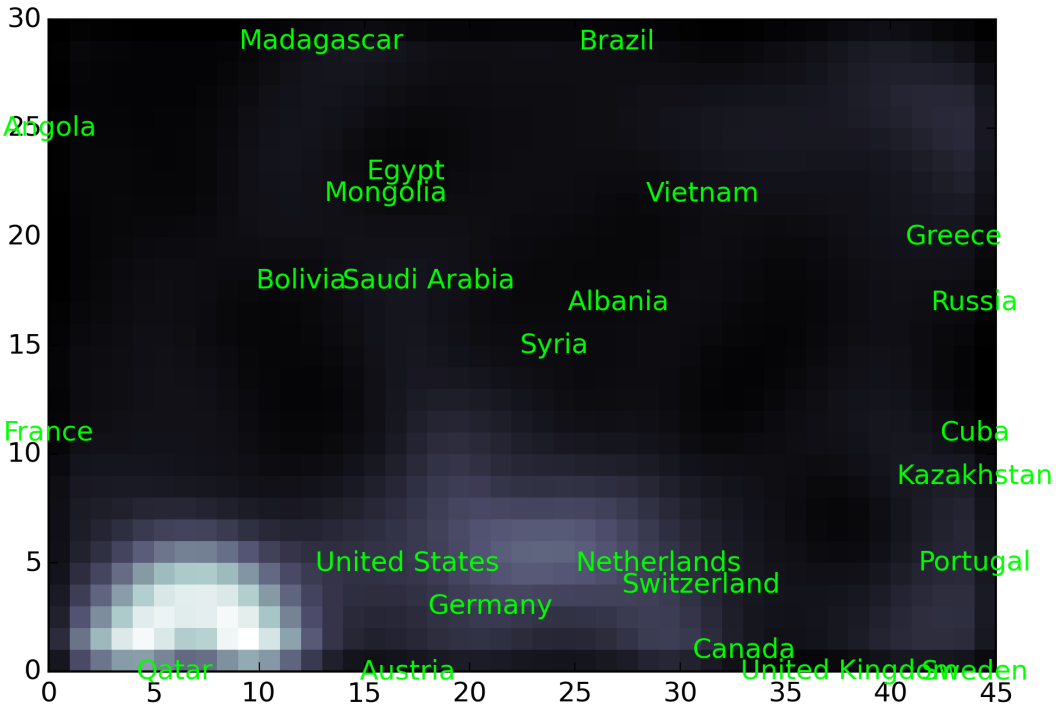

Figure 16.7: SOM with the projected planets.

16.5 Further Reading

1. PCA with Scikit-Learn: http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html
2. Wikipedia article on Self Organizing Maps: https://en.wikipedia.org/wiki/Self-organizing_map

Chapter 17

Introduction to Clustering

Introduction to clustering. K-means and k-medoids. Expectation-maximization.

17.1 Clustering

The goal of clustering is to group similar examples and separate different examples in different clusters. In other words, to create groups (clusters) of examples in a way that maximizes some measure of similarity between examples within the same cluster and minimizes the similarity between examples in different clusters. The term *clustering* can refer both to the process of computing the clusters and to the resulting set of clusters.

Clustering can help us understand the structure of the data and the relations between different examples and features. For one thing, grouping similar things together in categories that are distinct from other groups is an important part of how we understand the world around us. That is why we have words like "chair", "stool" or "sofa", that refer to clusters of objects. This is especially important when we have large datasets, as happens in fields like biology, astronomy or climatology, or when studying social networks online or credit card transactions. Clustering may also show us some important properties of the data. For example, clustering living organisms gives us an insight into their evolutionary relations. Figure 17.1 shows two examples of clustering used to understand the data. On the left panel, Darwin's "tree of life", an example of hierarchical clustering that gave an important insight into the mechanisms through which species originate. The right panel is an image from a study using a large set of positional data from Baidu, the most used search engine in China, to characterize empty neighbourhoods in chinese cities [6]. The authors used DBSCAN [10], a density-based clustering algorithm, to group the locations of Baidu users and estimate the home location of each user based on the density clusters of positional information. From this data, the authors then estimated the occupancy of residential neighbourhoods in chinese cities.

Clustering can also be used to *summarise* the data, replacing a large data set with a smaller number of data points that still retain the same overall structure. Figure 17.2 shows the result of using the *k-means* algorithm [16] to compute a simplified dataset, with a smaller number of points, but with the same "shape" as the original set.

There are several choices to make when deciding how to cluster some data. One problem is defining the number of clusters. Figure 17.3 illustrates this, showing the same set of points clustered into two, three or five clusters. Some clustering algorithms determine the number of clusters while others require

Figure 17.1: The left panel shows Darwin's depiction of the evolutionary relations between different organisms, an example of hierarchical clustering. The right panel is an image from [6], a characterization of "ghost cities" in China by density clustering.

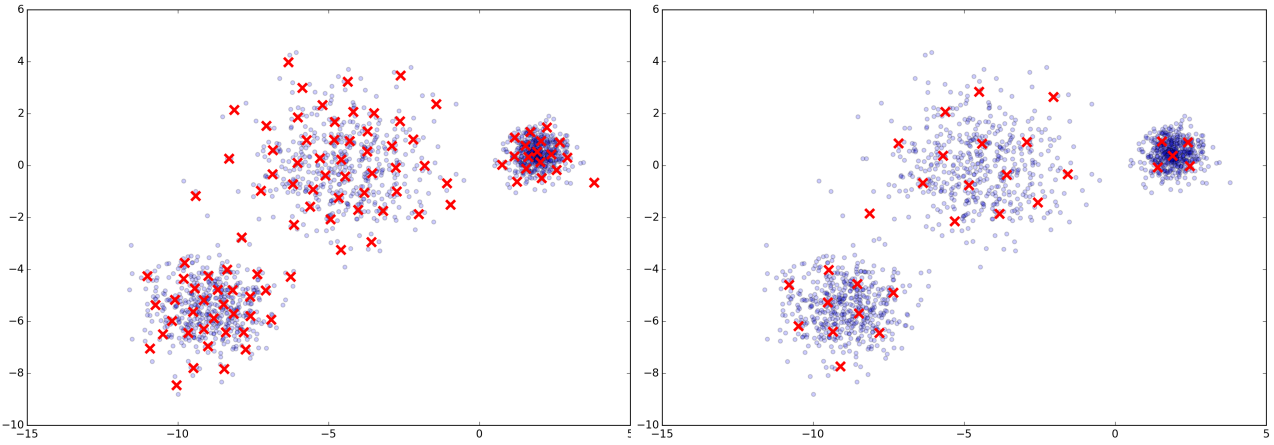

Figure 17.2: Summarizing the larger original dataset (blue points) into 100 or 30 points (red crosses), while still retaining the overall structure of the data.

that the number of clusters be specified in advance. We also need to decide if we want a *partitional* clustering, where the data set is divided into clusters at the same level, or a *hierarchical* clustering, where clusters are also clustered in higher-level clusters. Figure 17.4 shows the difference between partitional and hierarchical clustering.

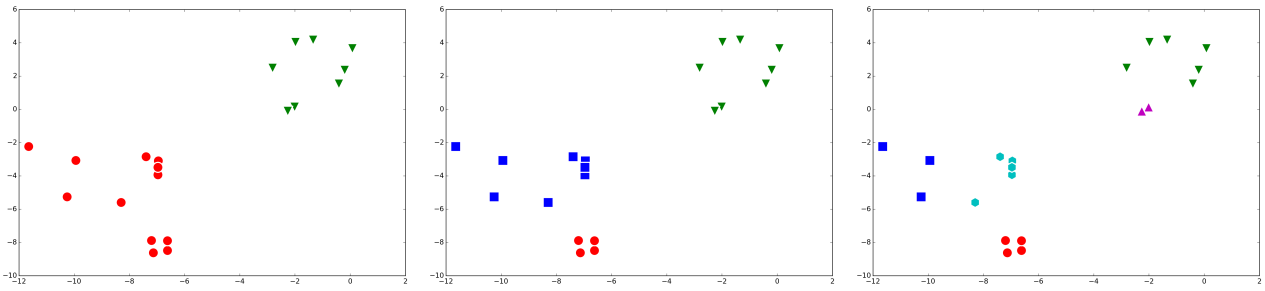

Figure 17.3: Choosing the number of clusters: two, three or five.

Regarding the membership of each example, the clustering can be *exclusive*, if each example belongs only to one cluster; *overlapping* if an example can belong to two or more clusters; *fuzzy* clustering

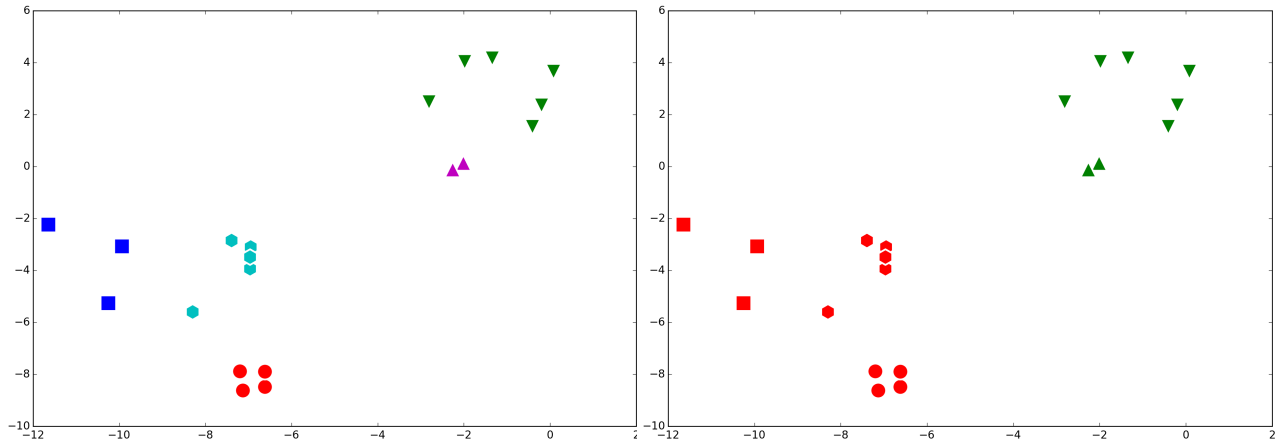

Figure 17.4: The left panel shows an example of partitional clustering. The right panel shows the same data set clustered hierarchically, with the colour representing the top-level clustering and the shapes the bottom level clustering. The lower level clusters are part of the higher level clusters.

if all examples belong to all clusters with some continuous membership value between 0 and 1; or *probabilistic* clustering if the membership value of each example to each cluster represents a probability. The clustering itself can be *partial* if not all examples belong to clusters or *complete* if all examples are assigned to clusters. Depending on the structure of the data and the clustering, the clusters can be *well-separated* if no example is more similar to any example outside its cluster than to any example within its cluster.

Clustering criteria can be based on different aspects of the structure of the data. Figure 17.5 shows some examples. Clustering based on *prototypes* assigns each example to the cluster represented by the closest prototype. With an euclidean distance measure, this results in a Voronoy partition of the feature space. *Contiguity-based* clustering creates clusters according to networks of contiguous examples, and *density-based* clustering assigns examples to clusters defined by high-density regions, allowing for some examples to be left unassigned and discarded as noise. Clustering can also be hierarchical, with groups clustered in larger groups, or defined by probability distributions, such as Gaussian Mixture Models, which we will cover later on.

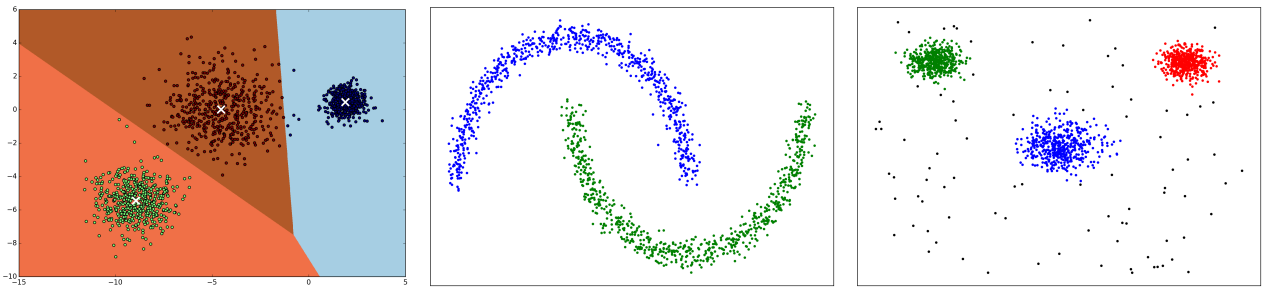

Figure 17.5: Examples of prototype-based clustering, contiguity-based clustering and density-based clustering.

17.2 K-means clustering

Lloyd's algorithm for k-means clustering algorithm[15] is conceptually very simple. It consists in dividing the data set into k clusters, each defined by the mean vector of the members of the cluster,

which is the *prototype* of that cluster. Each example belongs to the cluster represented by the closest prototype. Thus, k-means is an exclusive, partitional and prototype-based clustering method. The algorithm for computing the k-means clustering is:

1. Start with a random set of k prototypes.
2. Assign each example to the closest prototype.
3. Recompute each prototype as the mean point of all the examples assigned to that cluster.
4. Repeat steps 2 and 3 until convergence or some stopping criterion.

There are several possible initialization methods for k-means. For example, the *Forgy* method consists of assigning to each of the k prototypes the feature vector of a randomly selected examples as the starting point for the algorithm. The *Random Partition* method starts by randomly assigning each example to one of k clusters, and then computing the starting point of the prototypes as the mean point of each cluster. Figure 17.6 illustrates these two initialization methods, showing also how the *Random Partition* method tends to group the initial positions of the prototypes in the centre of the data space.

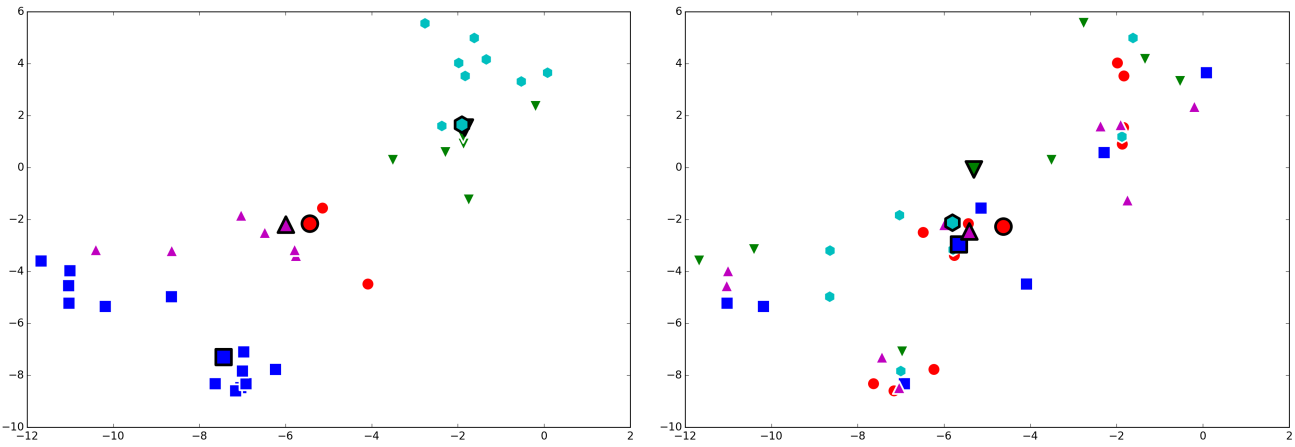

Figure 17.6: Initializing the k-means prototypes by the Forgy method (left panel) or the Random Partition method (right panel). The prototypes are represented with larger symbols and the smaller symbols represent the data points.

Figure 17.7 shows the first iterations of computing the clusters by the k-means algorithm, starting with the random (Forgy) assignment of the initial prototype positions and assignment of the data points to the clusters (left panel), recomputing the position of the cluster prototypes as the mean point of each cluster (middle panel) and then recomputing the cluster assignment by assigning each example to the cluster of the closest prototype (right panel).

To illustrate in more detail, we can see how to implement the k-means algorithm in Python. We start with a function that determines the cluster of each data point in matrix `data` given a matrix of centroid coordinates for the positions of the prototypes in `centroids`:

```
1 def closest_centroids(data,centroids):
2     ys = np.zeros(data.shape[0])
3     for ix in range(data.shape[0]):
4         dists = np.sum((centroids-data[ix,:])**2,axis=1)
5         ys[ix] = np.argmin(dists)
6     return ys
```

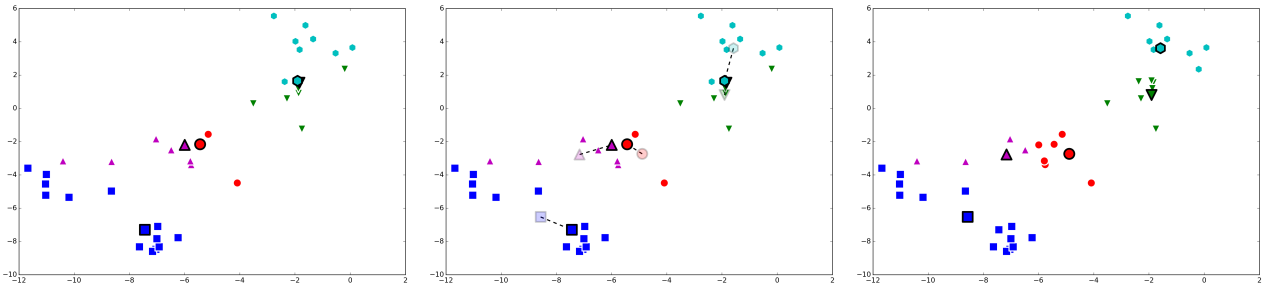

Figure 17.7: First iterations of the k-means algorithm. See text for more details.

Next, a function to recompute the centroids based on the assigned clusters. This function iterates through the lines of the `centroids` matrix computing, for each line, the means of the data points assigned to that cluster.

```
1 def adjust_centroids(data,centroids):
2     ys = closest_centroids(data,centroids)
3     for ix in range(centroids.shape[0]):
4         centroids[ix,:] = np.mean(data[ys==ix,:],axis=0)
```

Now we need a function for initializing the prototype positions (the centroids of the clusters). Using the Forgy method, we can do this assigning the coordinates of randomly selected examples. Note that the matrix returned is a copy of the selected lines of the data matrix, otherwise we could be returning pointers to the data points and altering the centroids would alter the data.

```
1 def forgy(data,k):
2     ixs = np.arange(data.shape[0])
3     np.random.shuffle(ixs)
4     return data[:k,:].copy()
```

Alternatively, we can initialize the centroids using the random partition algorithm.

```
1 def rand_part(data,k):
2     ys = np.random.randint(0,k,data.shape[0])
3     centroids = np.zeros((k,data.shape[1]))
4     for ix in range(k):
5         centroids[ix,:] = np.mean(data[ys==ix,:],axis=0)
6     return centroids,ys
```

To find the best prototypes for clusters we need a distance measure. Very often, the measure is the Euclidean distance. However, we can generalize this, as we saw before, with the Minkowsky distance, which depends on a parameter p:

$$D_{x,x'} = \sqrt[p]{\sum_d |x_d - x'_d|^p}$$

For $p = 2$ this is the Euclidean distance, and for $p = 1$ the Manhattan distance.

17.3 K-medoids

The *k-medoids* algorithm is a variant of the k-means algorithm with the difference that the prototypes always coincide with points in the data set. This makes it unnecessary to have a true distance measure,

since all we need is to measure similarities between points in the data set, and makes the algorithm more robust to noise and outliers. The k-medoids clustering can be computed using the *Partitioning Around Medoids* (PAM) algorithm, as follows:

1. Initialize the k prototypes (*e.g.* with the Forgy method).
2. Assign the examples to clusters.
3. For each medoid and each data point, test if swapping the medoid for that data point reduces the sum of pairwise dissimilarities between data points and respective medoids. If so, then update the medoid and reassign examples to clusters.
4. repeat step 3 until no improvement possible.

17.4 Expectation-Maximization

The *Expectation-Maximization* (EM) method is an important part of many unsupervised learning algorithms, and we shall revisit it in more detail in future chapters. But, for now, we can introduce it in a simplified overview and see how it relates to the k-means algorithm. Let us assume we have a set X of observed data — for example, the known data points — and a set Z of variables we do not observe, which are called *latent variables*. For example, the assignments of each data point to each cluster, which we initially do not know. We also have a set of parameters θ that we wish to adjust in order to maximize the likelihood of the parameters, which is the probability of all the data, including both the known and unknown variables, given the set of parameters θ. For example, the centroids representing the clusters.

$$L(\theta; X, Z) = p(X, Z|\theta)$$

We cannot compute this likelihood directly because we do not know the values Z. But we can estimate the posterior, or conditional, distribution of Z given the known X and some previous assumption about θ:

$$p(Z|X, \theta^{old})$$

This allows us to compute an expected value for Z and thus estimate the necessary parameters for the likelihood function $\mathcal{Q}(\theta, \theta^{old})$ for θ given some previous estimated θ^{old}. From the expected values of Z given X and θ^{old} and the likelihood of θ for the known X and the expected Z, we can write:

$$\mathcal{Q}(\theta, \theta^{old}) = \mathrm{E}_{Z|X,\theta^{old}} \ln p(X, Z|\theta)$$

We can now find the new values of $theta$ that maximize the likelihood function:

$$\theta^{new} = \arg\max_{\theta} \mathrm{E}_{Z|X,\theta^{old}} \ln p(X, Z|\theta)$$

Broadly speaking, this is what the k-means algorithm does[1]. The latent variables Z correspond to the assignment of examples to clusters and the known variables X correspond to the coordinates (features) of the examples in the data set. Given a set of prototype centroids, the θ^{old}, we can estimate the best values for Z by assigning each data point to the closest centroid. With this, we can obtain a

[1] K-means is not exactly Expectation-Maximization because k-means assigns each point to each cluster instead of estimating a probability of the point belonging to the cluster. However, we can see k-means as a limit case of EM.

maximum likelihood estimate of the centroid positions, the θ^{new}, by computing the mean point of each cluster.

Another way of understanding the EM algorithm is as an alternating sequence of optimizations with respect to different variables. This can be seen if we define the *distortion measure*:

$$J = \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} \|x_n - \mu_k\|^2$$

to evaluate the clusters, where r_{nk} is 1 if x_n belongs to μ_k, else 0. Thus, the r_{nk} variables are the latent variables assigning each point n to each cluster k. This distortion measure J, which is the sum of the squared distances between points and their respective cluster centroids, is what we want to minimize by optimizing the μ_k parameters, which are the coordinates of the centroids. However, we cannot do this without the r_{nk} values. Thus we first estimate the best r_{nk} values by minimizing J with respect to the r_{nk}, which consists simply of assigning each data point to the closest centroid, which results in the smallest distance added. Now we optimize the function J with respect to the μ_k centroids. Since J is a quadratic function on μ_k, the minimum can be found where the derivative is zero, which corresponds to the value of μ_k that is the mean point of the data points in cluster k.

As a limiting case of the more general Gaussian mixture models we will see in Chapter 20, the complete-data likelihood (including both the observed X and latent Z) for k-means tends towards:

$$\mathrm{E}_Z \left(\ln p(X, Z|\boldsymbol{\mu}) \right) \rightarrow -\frac{1}{2} \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} \|x_n - \mu_k\|^2 + C$$

Thus, in these conditions, EM corresponds to the alternating minimization of J we saw above. However, we will postpone a more detailed explanation of the EM method to Chapter 20, after looking at the Gaussian mixture models.

17.5 Application example for k-means

The k-means algorithm is often used for vector quantization, which is a procedure for reducing vectors in a range of values to a smaller set of representative prototypes. In this example, we'll use k-means to quantize the colour space of an image[2]. Figure 17.8 shows the starting image and the results of colour quantization with 64 and 8 centroids.

We start by loading the image and converting it into a set of two-dimensional vectors using the red and green colour components, since this image has no blue components. This is easy to do by using the `imread` function from the Scikit-image library:

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from sklearn.cluster import KMeans
4 from skimage.io import imsave,imread
5
6 rosa = imread("Rosa.png")
7 w,h,d = rosa.shape
8 cols = np.reshape(rosa/255.0, (w * h, d))
9 image_array = cols[:,:2]
```

[2]This example is based on the Scikit-learn demo on colour quantization, http://scikit-learn.org/stable/auto_examples/cluster/plot_color_quantization.html, using an example of vector quantization in Wikipedia, https://en.wikipedia.org/wiki/K-means_clustering

Figure 17.8: The original image (left panel) compared with two different colour quantizations, reducing the colour space to 6 bits (64 colours) and 3 bits (8 colours).

We can now plot all the pixels in the red-green colour space

```
plt.figure(figsize=(12,8))
plt.xlabel('Red')
plt.ylabel('Green')
plt.scatter(image_array[:, 0], image_array[:, 1], color=cols.tolist(), s=10)
plt.axis([0,1,0,1])
plt.savefig('L17-rosa-plot.png', dpi=200,bbox_inches='tight')
```

and use the k-means algorithm to find the best set of k prototypes to represent the colours. For example, for 64 colours, we can use the `KMeans` class from the `cluster` module of the Scikit-learn library. Computing the centroids and plotting over the plot of the set of points:

```
n_colors = 64
kmeans = KMeans(n_clusters=n_colors).fit(image_array)
labels = kmeans.predict(image_array)
centroids = kmeans.cluster_centers_
plt.scatter(centroids[:, 0], centroids[:, 1], marker='x', color='k',s=200, linewidths=5)
plt.scatter(centroids[:, 0], centroids[:, 1], marker='x', color='w',s=150, linewidths=2)
plt.savefig('L17-rosa-plot-cs-'+str(n_colors)+'.png', dpi=200,bbox_inches='tight')
```

The result is shown in Figure 17.9

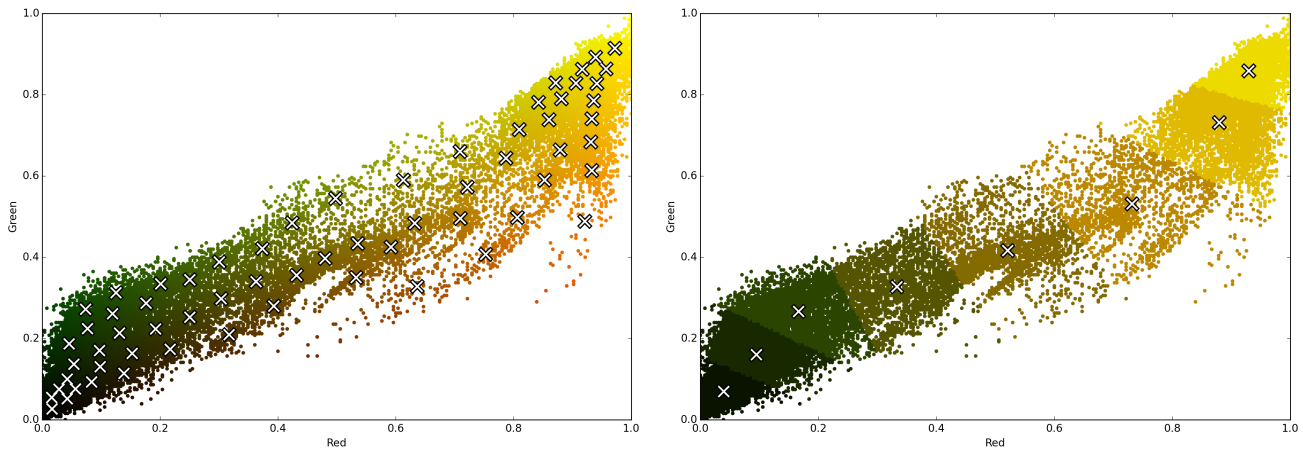

Figure 17.9: Quantizing the colour-space into 64 or 8 colours.

17.6 Further Reading

1. Alpaydin [2], Section 7.3
2. Marsland [17], Chapter 9
3. Bishop [4], Section 9.1

Chapter 18

Clustering: beyond prototypes

Affinity Propagation clustering and problems with prototype-based clustering. Density Clustering. Clustering validation.

18.1 Affinity Propagation clustering

Clustering using prototypes can be useful in some situations but inadequate in others. Because prototype-based clustering assigns data points to clusters according the their similarity to the prototypes, this type of clustering does not work well with clusters that are not globular or have different variances. Another difficulty arises if, like with k-means, we need to specify the number of clusters, which can lead to a poor clustering in some cases, although, in others, this can be a useful feature as we saw in the case of vector quantization, where the ability to specify the number of prototypes allows us to control the quantization. Figure 18.1 shows these aspects of prototype-based clustering.

Affinity Propagation solves the problem of having to pre-determine the number of clusters to generate. This algorithm is based, conceptually, on the idea of data points passing messages of "responsibility" sent by each data point to the candidates for cluster prototypes, indicating how suitable each candidate is according to that data point, and messages of "availability", sent by each prototype candidate to the data points indicating how adequate the candidate seems to be based on the support it has for being a prototype. Thus, we define:

- The similarity matrix $s_{i,k}$, indicating how alike two data points are, with $s_{k,k}$ indicating the propensity for being a prototype.
- Responsibility matrix $\mathbf{R}$, with $r_{i,k}$, indicating the suitability of k as prototype for i as estimated by i
- Availability matrix $\mathbf{A}$, with $a_{i,k}$, indicating the suitability of k to be prototype for i as estimated by k

The algorithm begins by initializing $\mathbf{R}$ and $\mathbf{A}$ to zero. Then assign to each value of $r_{i,k}$ the similarity between point i and prototype candidate k minus the largest sum of the affinity and similarity between i and any other prototype:

$$r_{i,k} \leftarrow s_{i,k} - \max_{k' \neq k} \left(a_{i,k'} + s_{i,k'} \right)$$

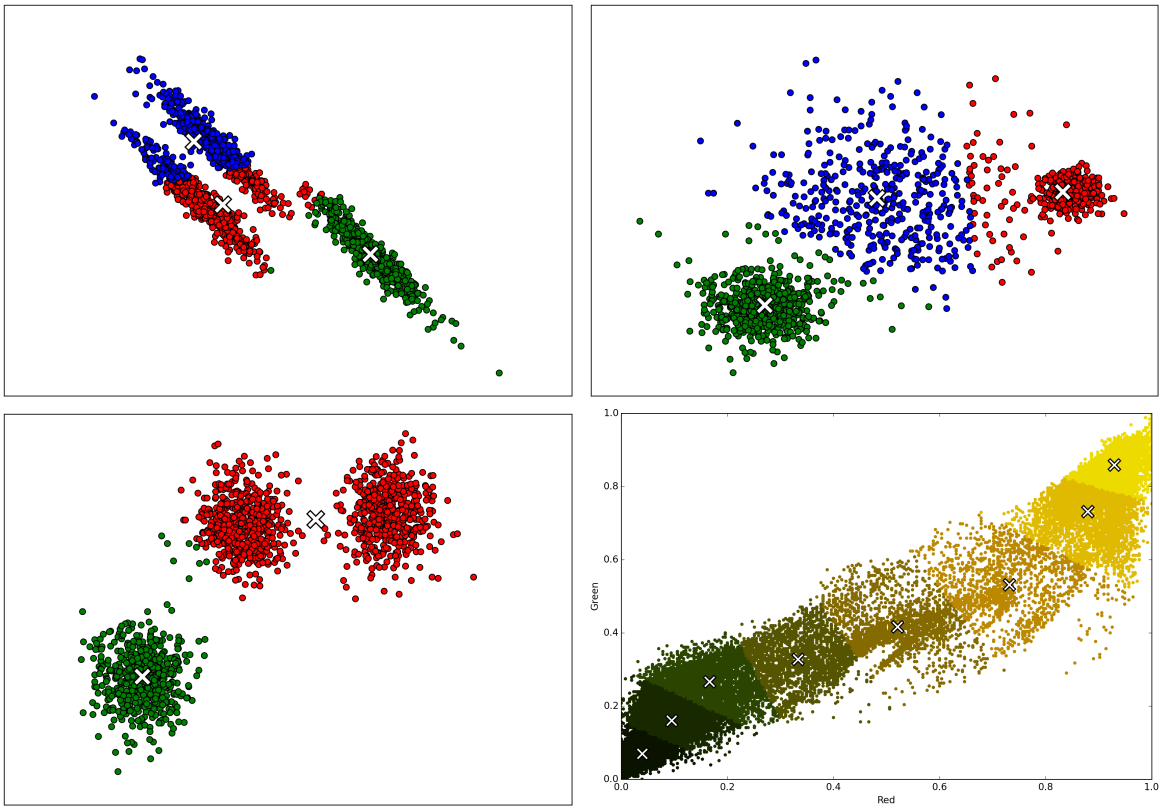

Figure 18.1: Prototype clustering fares poorly when the clusters are not globular (top-left panel), have different variances (top-right) or the number of prototypes chosen is incorrect (bottom-left). However, these features may be useful in some cases, such as in vector quantization (bottom-right).

In the first iteration, with $\mathbf{A}$ set to zero, this is simply the similarity between i and k. After the first iteration, this also discounts the availability of the other candidates as prototypes for point i, the values of $a_{i,k'}$, which are negative numbers. So, basically, the responsibility sent from i to k will depend on how better than other candidates, in similarity and availability, k seems to be for i. In other words, candidate prototypes are competing for the votes of the data points.

Then the algorithm updates the $\mathbf{A}$ matrix by considering the self responsibility of k and the votes k gets from other data points.

$$a_{i,k(i\neq k)} \leftarrow \min\left(0, r_{k,k} + \sum_{i' \notin \{i,k\}} \max(0, r_{i',k})\right)$$

The self availability is updated as follows, serving as evidence that k is a prototype:

$$a_{k,k} \leftarrow \sum_{i' \neq k} \max\left(0, r_{i',k}\right)$$

To identify prototypes, at each iteration, the algorithm computes for each i the k' point with the largest sum of availability and responsibility:

$$k' = \arg\max_{k} a_{i,k} + r_{i,k}$$

If $k' = i$, then i is a prototype of a cluster. Otherwise, i belongs to the cluster with prototype k'. Figure 18.2 shows some stages of the affinity propagation algorithm. Initially, nearly all points consider

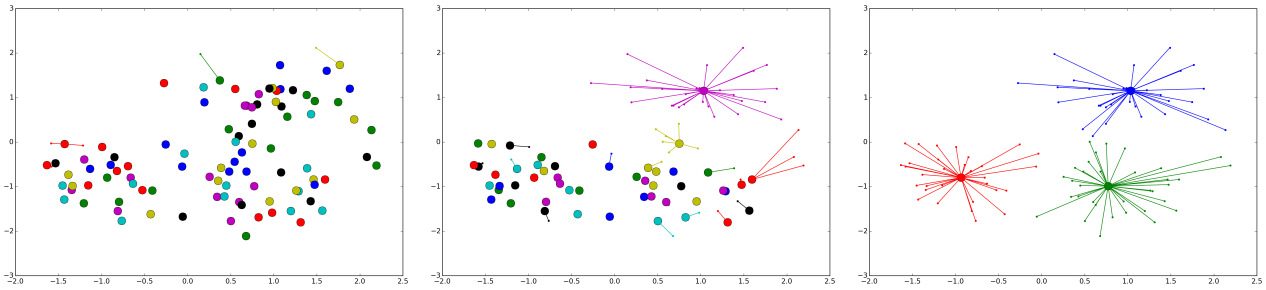

Figure 18.2: Stages of training in affinity propagation clustering.

themselves the best prototype for their own clusters. But as availability and responsibility is propagated, more votes will accumulate with some candidates, becoming prototypes for larger groups of points.

This solves the problem of determining the number of clusters, since these result automatically from the algorithm and depend on the structure of the data. However, it does not solve other problems with prototype-based clustering that stem from attributing points to clusters based on the similarity to the cluster prototypes. Figure 18.3 illustrates this problem. Since all it matters is the similarity to the prototype, and all points must be long to the cluster with the closest prototype, prototype-based clustering is unable to account for some relations between the points.

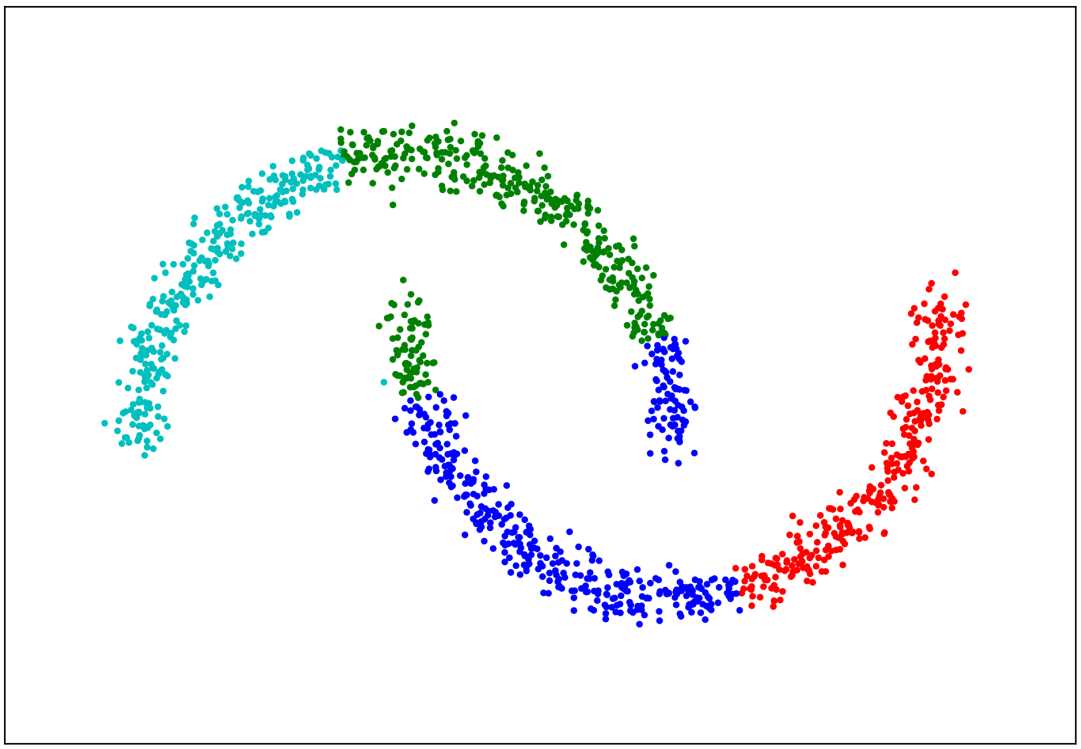

Figure 18.3: Affinity propagation does not solve all problems with prototype-based clustering.

Affinity propagation clustering in Scikit-learn can be done with the `AffinityPropagation` class in the `cluster` module.

18.2 Density-based clustering

The *Density-based spatial clustering of applications with noise* (DBSCAN) algorithm[10] takes a different approach, which solves these problems with prototype-based clustering. Defining the ϵ

neighbourhood N_ϵ of each point as the set of points within distance ϵ, a point p is a *core point* if the number of points in the N_ϵ of p is at least equal to a parameter `minPts`. A point q is *reachable* from p if p is a core point and q is in the neighbourhood N_ϵ of p or in the neighbourhood N_ϵ of any core point that is reachable from p. This recursive definition means that q is reachable from p if there is a path of reachable core points from p to q. The DBSCAN algorithm proceeds as follows. For each point p, if the number of points in the neighbourhood N_ϵ of p is less than `minPts`, p is presumed to be noise. Otherwise, a cluster is created for p, which is a core point, and all neighbours of p are added to the cluster. If any neighbour of p is a core point belonging to another cluster, the clusters are merged.

This algorithm solves the problem of the shape of the clusters that arises with prototype clustering, since the cluster membership propagates along the paths of nearby core points. Figure 18.4 shows the result of clustering with the DBSCAN algorithm. The blue and green points are assigned to the two different clusters, while the black points are considered noise and not assigned to any cluster.

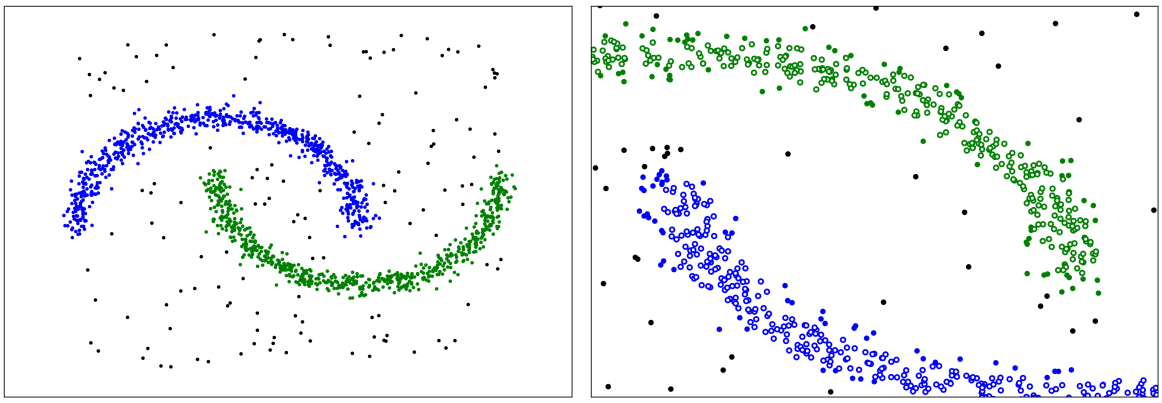

Figure 18.4: DBSCAN clustering. The right panel shows the core points (white centers), the non core points assigned to clusters (filled, green and blue) and points not assigned to clusters, which are considered noise, in low density regions (black).

The `cluster` module of the Scikit-Learn library offers a `DBSCAN` class for clustering with this algorithm.

18.3 Clustering Validation

In supervised learning we can always evaluate the results by measuring the error between the predictions and the data labels. But with clustering we may not have that possibility, if we use unlabelled data. So we may need to assess clusters by some measure of the "goodness" of the clustering. This may be necessary in different contexts, such as to determine if the data has structure, if the right number of clusters were predicted, if the clusters fit the data structure and if they fit some external information, such as class labels, if such information is available. It may also be necessary to evaluate clustering to compare different sets of clusters. Note that, in the following subsections, the similarity measure can be substituted by a dissimilarity or distance measure. The only difference is that the meaning of lower or higher scores will be reversed, as a high score using a similarity measure will, for the same clusters, be a low score using a distance measure, and the quality of the clusters may be proportional or inversely proportional to the score depending on the functions used.

Cluster cohesion

Cluster cohesion is a measure of the similarity of points within each cluster. For each cluster, the cluster cohesion score is:

$$cohesion(C_i) = \sum_{x \in C_i} \sum_{y \in C_i} S(x, y)$$

For prototype-based clusters, the distance or similarity can be measured with respect to the prototype of each cluster:

$$cohesion(C_i) = \sum_{x \in C_i} S(x, \mathbf{c}_i)$$

Cluster separation

Cluster separation measures the similarity, or dissimilarity, of examples between different clusters, summing the similarity measure between all pairs of points in different clusters.

$$separation(C_i, C_j) = \sum_{x \in C_i} \sum_{y \in C_j} S(x, y)$$

For prototype-based clusters, the distance or similarity can be measured with respect to the prototype of each cluster:

$$separation(C_i, C_j) = S(\mathbf{c}_i, \mathbf{c}_j)$$

Sum of Squared Errors

For prototype-based clustering, we can define a sum of squared errors as the sum of the distance to the prototype or, more often, the sum of the squared distance.

$$SSE = \sum_{k} \sum_{x \in C_k} dist(x, \mathbf{c}_k)$$

Silhouette Score

Given $a(i)$ as the average distance between point i and all other points in the same cluster and $b(i)$ as the average distance of point i to all points in the nearest cluster, with the nearest cluster being the one with the smallest average distance to i, the silhouette score for point i is the fraction:

$$s(i) = \frac{b(i) - a(i)}{\max\left(a\left(i\right), b\left(i\right)\right)}$$

Averaging over all points, we can obtain the silhouette score for the clustering, a value ranging from -1 to 1 with a higher value the more the distance between clusters, $b(i)$, is greater than $a(i)$. The silhouette score is available on the Scikit-Learn library as the `silhouette_score` function in the `metrics` module. Figure **??** shows the silhouette score for different clusterings using the k-means algorithm with different numbers of clusters and data sets.

When working with labelled data, in supervised learning, we can also evaluate the clusters by comparing cluster to the known structure of the data. While the previous scores we saw depend on *internal indices*, computed solely from the clusters and the structure of the data, with labelled data it is possible to use this information to obtain *external indices* for evaluating clusters.

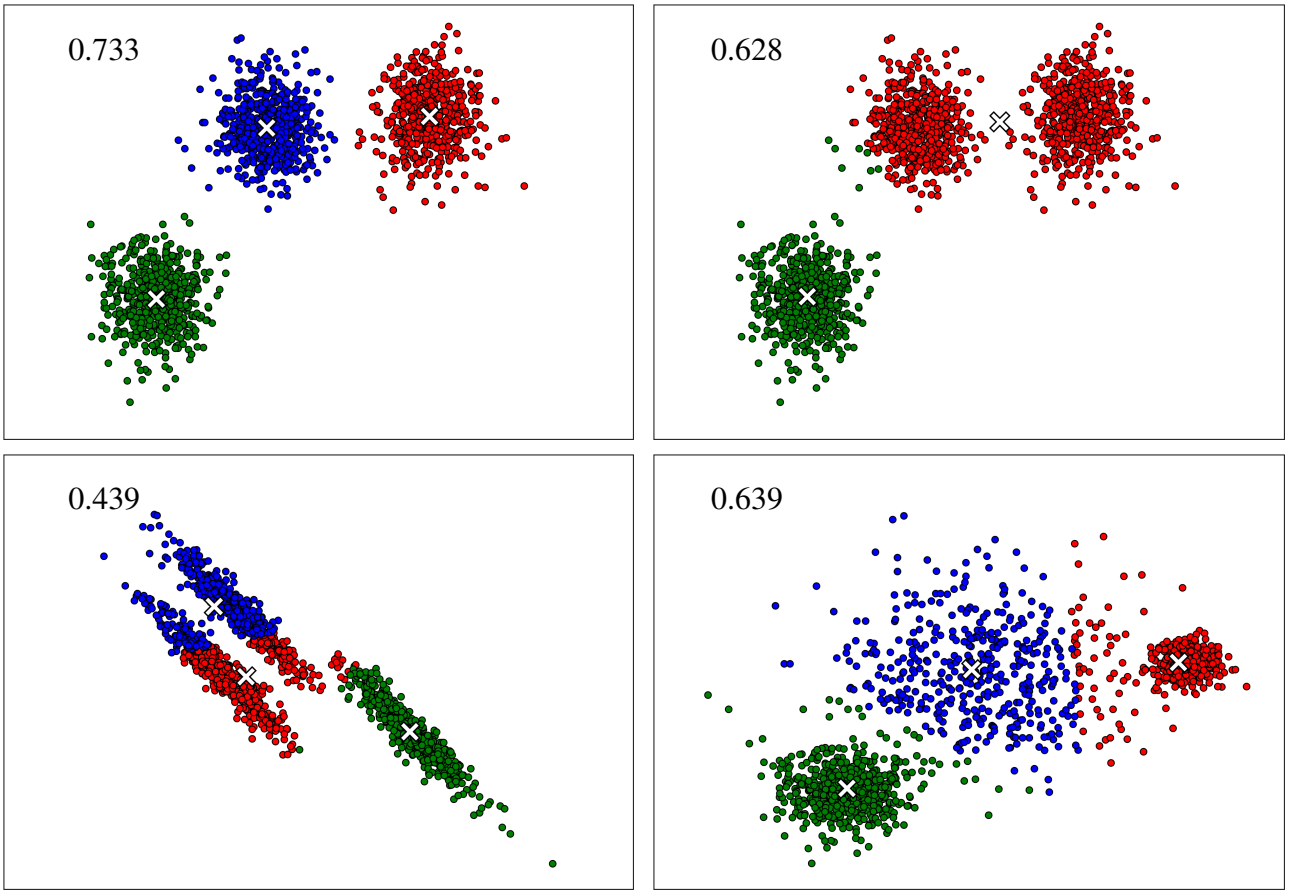

Figure 18.5: Silhouette score for different clusterings.

18.4 Further Reading

1. Frey *et. al.*, Clustering by passing messages between data points [11]
2. Ester *et. al.*, A density-based algorithm for discovering clusters in large spatial databases with noise. [10]
3. Scikit-learn documentation on clustering:`http://scikit-learn.org/stable/modules/clustering.html`

Chapter 19

Hierarchical clustering

Hierarchical Clustering. Agglomerative and Divisive Clustering. Clustering Features.

19.1 Hierarchical clustering

Deciding the best number of clusters is often difficult, as the structure of the data may not provide an obvious solution for this problem. For example, if we want to cluster all living organisms, it is not clear how many clusters we should have. In this case, the reason is that living organisms are related in a family tree, in a range of degrees of distance. The best option is to represent this structure in a series of nested clusters, and clusters of clusters, and so on. This is done with *hierarchical clustering*. Figure 19.1 shows two examples of hierarchical clustering. The tree of life, a hierarchical clustering of living species that also represents their evolutionary relations, and hierarchical clustering for analysing similarities in gene expression patterns in different organisms.

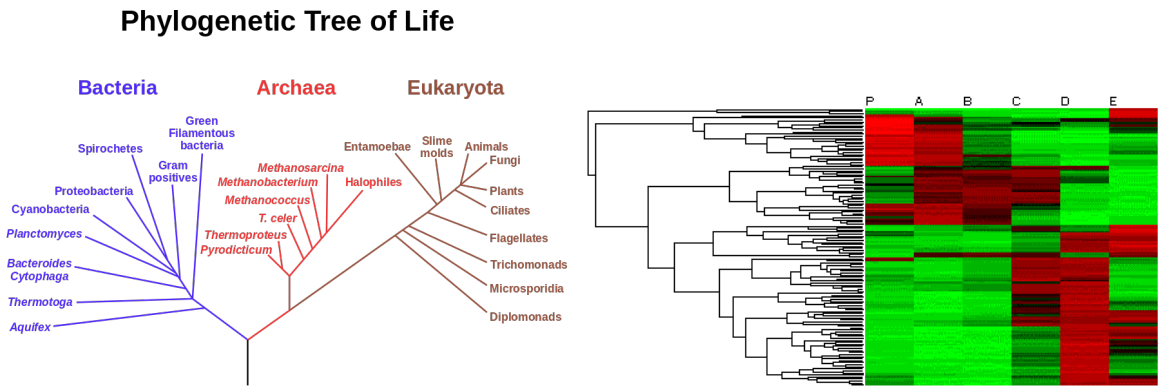

Figure 19.1: Examples of hierarchical clustering. Left panel, hierarchical clustering of living organisms, indicating evolutionary relations. Image source: Wikipedia. On the right panel, hierarchical clustering of gene expression data (Mulvey and Gingold, Online Computational Biology Textbook).

A hierarchical clustering can be represented as a dendrogram (a tree) by joining together first the examples that are more similar and then gradually joining the most similar clusters until all links are found, as shown in Figure 19.2. This means that we need to define how to measure the similarity, or dissimilarity, between examples in our dataset but also how to measure similarity between clusters of examples, because we need to decide how to cluster clusters into sets of larger clusters.

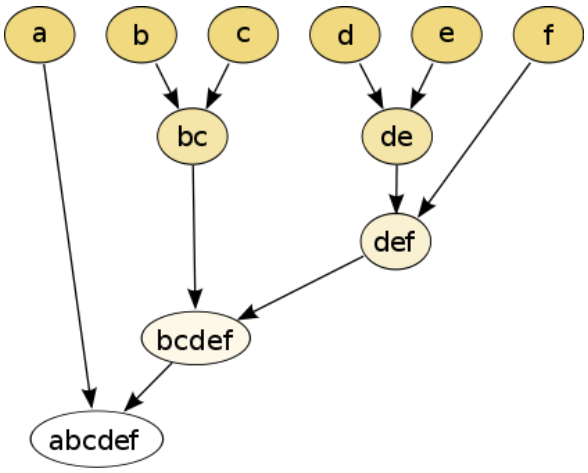

Figure 19.2: Hierarchical clustering represented as a dendrogram. Image source: Wikipedia.

There are several ways of thinking about this problem. We can think about *proximity* between examples as a generic term of "likeness", without any precise definition. *Similarity*is more well defined, generally a number between 0 and 1 that indicates how alike examples are. *Dissimilarity* is also a quantitative measure, in this case of difference between examples, and *distance* is a special case of a dissimilarity measure that respects the algebraic properties of a distance. Namely, not negative, symmetrical and respecting the triangle inequality:

$$d(x, y) \geq 0 \ , \ d(x, y) = d(y, x) \ , \ d(x, z) \leq d(x, y) + d(y, x)$$

There are many possible distance measures. Some of the most used are Euclidean, Manhattan and squared Euclidean distance.

- Euclidean: $\|x - y\|_2 = \sqrt{\sum_d (x_d - y_d)^2}$
- Squared Euclidean: $\|x - y\|_2^2 = \sum_d (x_d - y_d)^2$
- Manhattan: $\|x - y\|_1 = \sum_d (x_d - y_d)^2$
- Mahalanobis (normalized by variance): $\sqrt{(x - y)^T Cov^{-1}(x - y)}$

For strings and sequences in general, some useful measures are the Hamming distance, which is the count of differences between the strings, or the Levenshtein distance, or edit distance, counting the number of single-character edits (insertions, deletions or substitutions) needed to transform one string into the other.

Apart from a way to measure similarity or distance between examples, we must also measure distance between clusters. The method for evaluating cluster distance is the *linkage*, and there are also several ways of doing this.

- *Single linkage*: distance between clusters is the distance between the closest points.

$$dist(C_j, C_k) = min\left(dist(x \in C_j, y \in C_k)\right)$$

- *Complete linkage*: distance between the most distant points.

$$dist(C_j, C_k) = max\left(dist(x \in C_j, y \in C_k)\right)$$

- *Centroid linkage*: distance between the centroids of the two clusters.

$$dist(C_j, C_k) = dist\left(\frac{\sum x \in C_j}{|C_j|}, \frac{\sum y \in C_k}{|C_k|}\right)$$

- *Average linkage*: average distance between all pairs of points from the different clusters.

$$dist(C_j, C_k) = mean\left(dist(x \in C_j, y \in C_k)\right)$$

- *Median linkage*: median distance between all pairs of points from the different clusters.

$$dist(C_j, C_k) = median\left(dist(x \in C_j, y \in C_k)\right)$$

- *Ward linkage*: join clusters that minimize Sum of Squares Error:

$$\sum_{n=1}^{N}\sum_{k=1}^{K} r_{nk}\|x_n - \mu_k\|^2$$

Figure 19-linkage illustrates some examples of linkage methods.

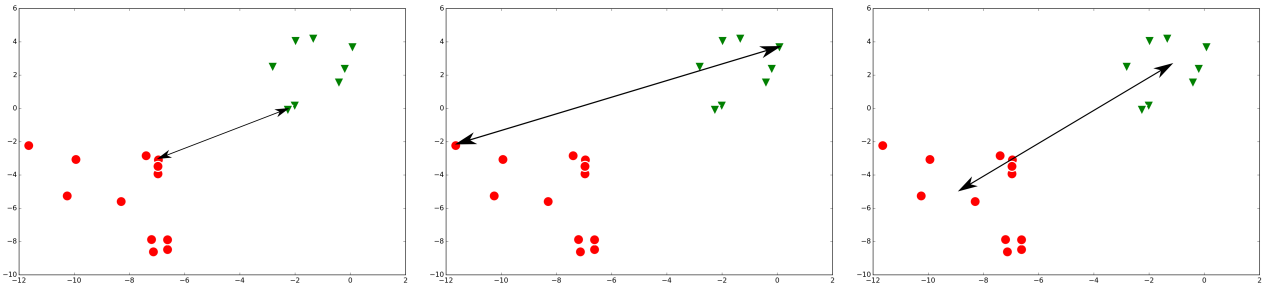

Figure 19.3: Single, complete and centroid linkage methods.

The obvious advantages of hierarchical clustering is avoiding the need to specify a number of clusters, both before or after clustering, and the possibility of revealing some hierarchical structure in the data. The disadvantages are that hierarchical clustering must generally be done in a single pass, with a greedy algorithm, which may introduce errors, and if the hierarchical structure assumed by this type of clustering does not exist in the data the result may be confusing or misleading.

Agglomerative clustering is a bottom-up approach that begins with singleton clusters and repeatedly joins the best two clusters, according to the linkage method used, into a higher level cluster until all elements are joined. The time complexity of agglomerative clustering is generally $O(n^3)$, but can be improved with linkage constraints.

Divisive clustering is a top-down approach that begins with a single cluster containing all examples and iteratively picks a cluster to split and separates it into smaller clusters until some number of clusters is reached. The theoretical time complexity for divisive clustering is $O(2^n)$ for an exhaustive search and this approach needs an additional clustering algorithm for splitting each cluster. However, the time complexity in practice can be lower, depending on the clustering algorithm used, and it may be better than agglomerative clustering if we only want a few levels of hierarchical clustering.

19.2 Hierarchical to partitional

Although a hierarchical clustering is a tree of clusters inside other clusters, we can convert it into a partitional clustering, with a set of clusters at the same level, by cutting some arcs of the tree. The farther we go from the root of the tree, the greater the number of clusters generated. Figure ?? illustrates this process.

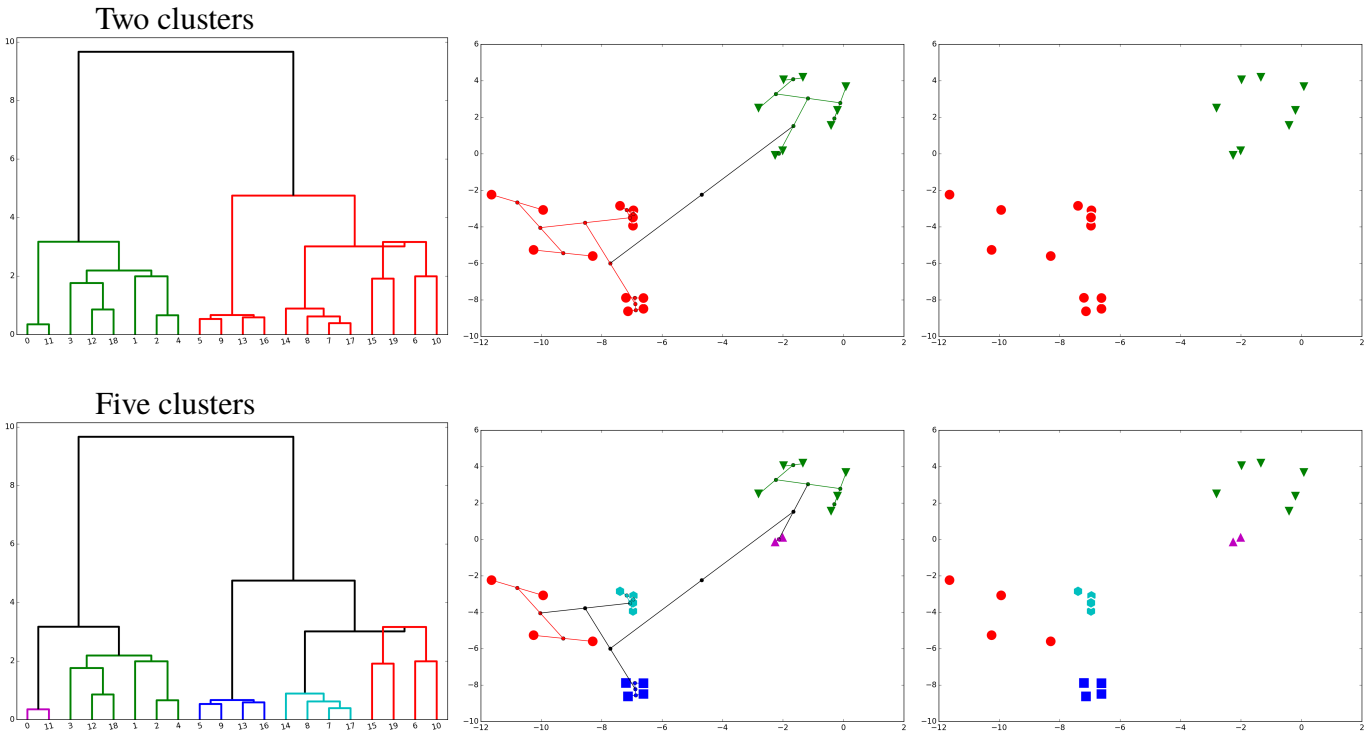

Figure 19.4: Partitioning a hierarchical clustering by cutting the tree at the desired level.

19.3 Connectivity constraints

In agglomerative clustering, we can restrict which clusters to join by adding connectivity constraints. These constraints specify which examples are considered connected and only clusters with connected examples, from one cluster to the other, can be joined into larger clusters. This helps solve some problems like Figure 19.5 illustrates. The left panel shows the result of agglomerative clustering without connectivity constraints. Since the linkage method used (Ward) takes into account only distances between the points, in order to minimize the SSE, the clusters include examples across the gap separating different stretches of the "ribbon" in which the data is structured. A connectivity constraint that restricts the connection of each example only to the 10 nearest neighbours creates a graph of connections that respects the structure of the data and prevents these inadequate clusters from forming.

To create this matrix with the connectivity constraints, we can use the `kneighbors_graph` function from the `neighbors` module of the Scikit-learn, and then use the connectivity constraints matrix in the `AgglomerativeClustering` class, as shown below. The result is shown in the right panel of Figure 19.5.

```
1 from sklearn.cluster import AgglomerativeClustering
2 from sklearn.neighbors import kneighbors_graph
```

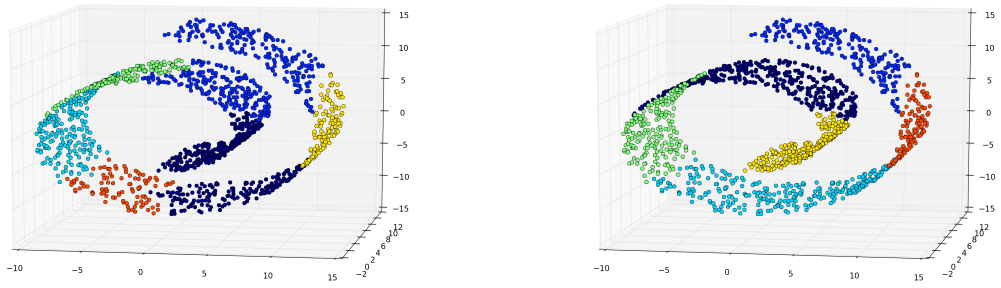

Figure 19.5: Agglomerative clustering with Ward linkage, without connectivity constraints (left panel) and with connectivity constraints connecting only the 10 nearest neighbours of each example.

```
3
4 connectivity = kneighbors_graph(X, n_neighbors=10, include_self=False)
5 ward = AgglomerativeClustering(n_clusters=6, connectivity=connectivity,
6                                linkage='ward').fit(X)
```

19.4 Choosing the linkage method

Scikit-Learn currently offers three linkage methods for agglomerative clustering: complete, average and Ward linkage. Figure 19.6 shows an example data set clustered to three clusters using agglomerative clustering and the three linkage methods. Complete linkage tends to favour larger clusters, so leads to a poor relation between the clusters and the data structure in some cases, as the figure shows (left panel). Average linkage is better, in these cases (middle panel), and Ward linkage (right panel), minimizing the SSE measured in the clusters, seems to work best. However, Ward linkage can only be used when the dissimilarity measure is the Euclidean distance, so if another measure must be used average linkage tends to be the best option.

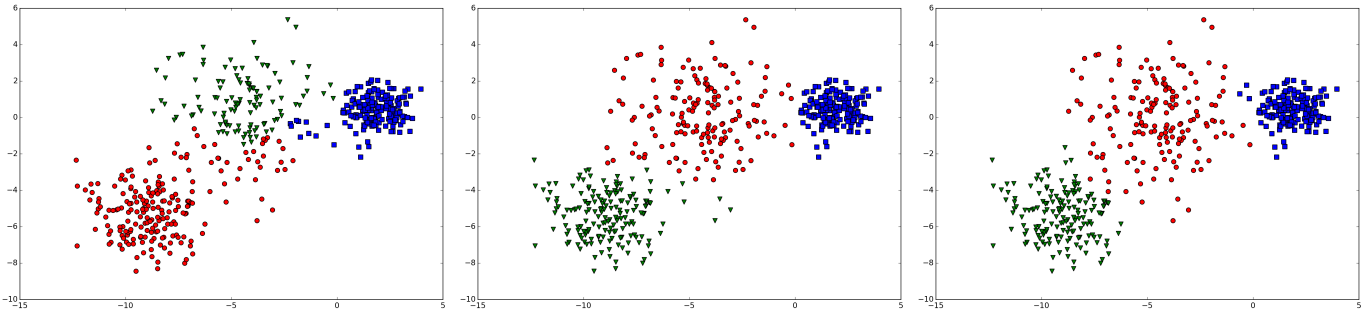

Figure 19.6: Agglomerative clustering of the same data set with (left to right) complete, average and Ward linkage.

19.5 Bisecting k-means

An example of a divisive hierarchical clustering algorithm is the *bisecting k-means*. The algorithm is:

1. Start with all the examples in a single cluster.

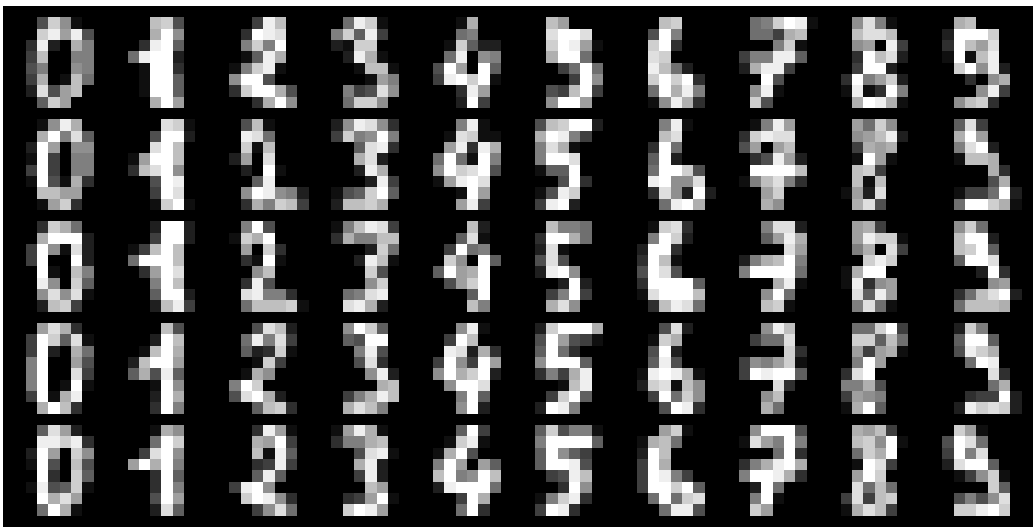

Figure 19.7: Some examples from the handwritten digits dataset.

2. Choose the best cluster for splitting (*e.g.* the largest or the one with the lowest score).
3. Split the best candidate with k-means, using $k = 2$.
4. Repeat steps 2 and 3 until the desired number of clusters is reached.

Although the time complexity for an exhaustive search in divisive clustering is $O(2^n)$, using the k-means algorithm reduces the complexity (although at the cost of having a more greedy divisive clustering) and the possibility of stopping at the desired level may make this algorithm preferable to agglomerative clustering in some cases, since agglomerative clustering must run until the complete tree is generated.

19.6 Clustering features

Conceptually, clustering features is the same as clustering examples. We need but imagine that we transpose the matrix with all examples in rows and features in columns and obtain a new matrix were the examples are in the columns, and are now considered features, and the original features, now in rows, are examples. Clustering features allows us to identify similar features and reduce the dimensionality of the data by grouping these together in a single feature. With hierarchical clustering we have a simple way of controlling how many groups of features we use and thus the dimensionally of the resulting data set.

To illustrate this, we will simplify the handwritten digits data set, which consists of digitized handwritten digits into grayscale bitmaps of 64 pixels. Figure 19.7 shows these data.

The data set has a total of 1797 examples with 64 features each so, for feature clustering, we will convert it into a set of 64 examples each with 1797 features. Then we cluster it into 16 clusters, corresponding to 16 features in the original data set, which we can extract by averaging all features in each cluster. We also add a connectivity constraint to restrict clustering to neighbouring pixels in the original image. Feature clustering is done automatically in the `FeatureAgglomeration` class, so the complete code, including loading the data set, is simply:

```
1 import numpy as np
2 from sklearn import datasets, cluster
```

```
3 from sklearn.feature_extraction.image import grid_to_graph
4
5 digits = datasets.load_digits()
6 images = digits.images
7 X = np.reshape(images, (len(images), -1))
8 connectivity = grid_to_graph(images[0].shape[0],images[0].shape[1])
9 agglo = cluster.FeatureAgglomeration(connectivity=connectivity, n_clusters=16)
10 agglo.fit(X)
11 X_reduced = agglo.transform(X)
12 X_restored = agglo.inverse_transform(X_reduced)
```

Lines 5-7 are for loading the data and converting the image matrices into a matrix of examples (rows) and features (columns). Line 8 is for creating the connectivity matrix with the neighbours of each pixel in the 64×64 image matrix. Lines 9 and 10 create the agglomerative clusterer and fit the data, and the last two lines complete the reduced dataset, with only 16 features, and a 64 features dataset with the feature values aggregated, averaging the features in the same cluster. Figure 19.8 shows the result. Although the digits in the reduced dataset are no longer recognizable as digits, it is easy to see that the patterns are different from digit to digit, so this process reduced the number of features without losing much information.

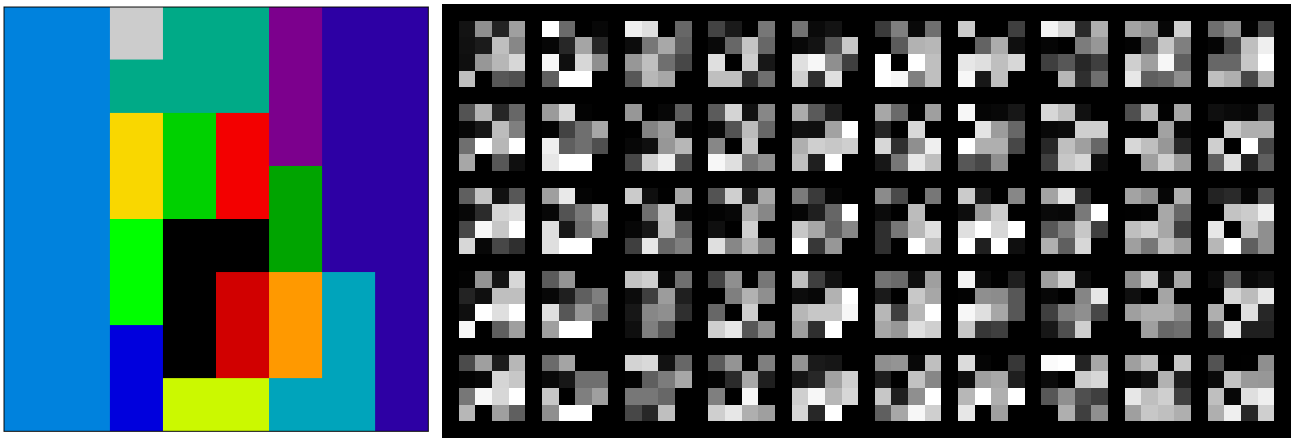

Figure 19.8: Feature clustering. The original handwritten digits features were clustered as shown in the left panel. Using only these 16 clusters as 16 features, the reduced data set is illustrated on the right panel.

19.7 Further Reading

1. Scikit-learn documentation on clustering:http://scikit-learn.org/stable/modules/clustering.html

2. Alpaydin [2], Section 7.7

Chapter 20

Probabilistic clustering

Fuzzy sets and clustering. Fuzzy c-means. Probabilistic Clustering: mixture models. Expectation-Maximization revisited.

20.1 Fuzzy Clustering

In conventional set theory, elements either belong or do not belong to a set. In such case, we are dealing with *crisp* sets. In *fuzzy* set theory, each element x has a membership value $u_S(x) \in [0, 1]$ specifying by how much x belongs to set S. Thus, a *fuzzy set* S is a set of ordered pairs of elements and their respective membership function values:

$$S = \{(\mathbf{x}, u_S(\mathbf{x}))|\mathbf{x} \in X\}$$

This makes it possible to model different types of uncertainty, such as linguistic or categorical uncertainty, when we are unable to define exactly what we mean by some term or category. For example, the temperature at which something stops being cold and becomes warm or hot is not a precise (*crisp*) value. One way to account for this is to allow the membership of each temperature value to each category cold, warm or hot to vary continuously, as Figure 20.1 illustrates.

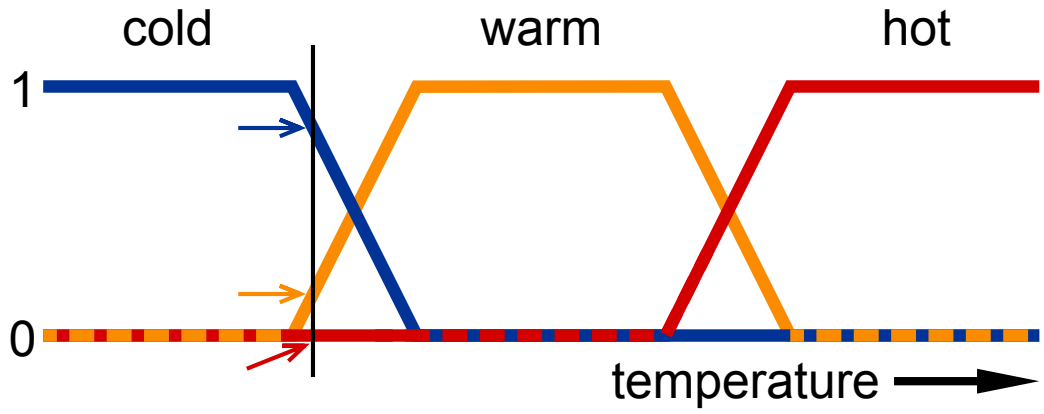

Figure 20.1: Fuzzy sets for cold, warm and hot temperatures, and respective membership values. Wikimedia, CC BY-SA 3.0 fullofstars

Fuzzy sets can also model uncertainty about information or predictions, but fuzzy membership is different from probability estimates. Conceptually, fuzzy membership is a measure of similarity to

some imprecise properties that characterize the set the element may belong to with a smaller or larger membership value, while probability is a measure either of a frequency of random events in the limit of infinite trials (frequentist interpretation) or of uncertainty but under precise definitions of concepts (Bayesian interpretation).

Fuzzy clustering rests on the notion of a *fuzzy c-partittion*. $\mathbf{U}(\mathbf{X})$ is a fuzzy c-partition of X if these three conditions hold. First, the membership values of all elements are between 0 and 1:

$$0 \leq u_k(\mathbf{x}_n) \leq 1 \qquad \forall k, n$$

Second, the total membership of each element to all c partitions is equal to 1:

$$\sum_{k=1}^{c} u_k(\mathbf{x}_n) = 1 \qquad \forall n$$

Finally, the total membership in each of the c partitions is between 0 and the total number of elements:

$$0 \leq \sum_{n=1}^{N} u_k(\mathbf{x}_n) \leq N \qquad \forall k$$

The *fuzzy c-means* algorithm is a clustering algorithm that finds a *fuzzy c-partittion* for the elements to cluster, with each partition being a cluster. From a set X of N data points, the algorithm returns the $c \times N$ membership matrix $u_k(\mathbf{x}_n)$, defining a fuzzy c-partition of X and determining the membership value of each element $\mathbf{x}_n, n \in \{1, ..., N\}$ to each cluster $k \in \{1, ..., c\}$. The *fuzzy c-means* algorithm also returns the set $\{C_1, ..., C_c\}$ of centroids of the partitions (clusters). These are found by minimizing the following squared error loss function:

$$J_m(X, C) = \sum_{k=1}^{c} \sum_{n=1}^{N} u_k(\mathbf{x}_n)^m \|\mathbf{x}_n - \mathbf{c}_k\|^2 \qquad m \geq 1$$

and subject to the constraint

$$\sum_{k=1}^{c} u_k(\mathbf{x}_n) = 1 \qquad \forall n$$

</p> The parameter m, typically $m = 2$, is the *degree of fuzzification*. The derivative of the loss function with respect to the membership values is zero at the points:

$$u_k(\mathbf{x}_n) = \frac{\left(\frac{1}{\|\mathbf{x}_n - \mathbf{c}_k\|^2}\right)^{\frac{2}{m-1}}}{\sum_{j=1}^{c} \left(\frac{1}{\|\mathbf{x}_n - \mathbf{c}_j\|^2}\right)^{\frac{2}{m-1}}}$$

and with respect to the centroids C_k:

$$C_k = \frac{\sum_{n=1}^{N} u_k(\mathbf{x}_n)^m \mathbf{x}_n}{\sum_{n=1}^{N} u_k(\mathbf{x}_n)^m}$$

That is, each centroid C_k is the weighted mean of the example vectors using the membership values. This algorithm is similar to the k-means algorithm, but using a continuous membership function instead of the $0, 1$ membership of crisp sets.

Like k-means, the fuzzy c-means algorithm also uses the Expectation-Maximization method. First, the expected value of the latent variables, the membership values, are computed from a random initial set of centroids $\{C_1, ..., C_c\}$:

$$u_k(\mathbf{x}_n) = \frac{\left(\frac{1}{\|\mathbf{x}_n - \mathbf{c}_k\|^2}\right)^{\frac{2}{m-1}}}{\sum_{j=1}^{c}\left(\frac{1}{\|\mathbf{x}_n - \mathbf{c}_j\|^2}\right)^{\frac{2}{m-1}}}$$

This, in turn, allows the update of the centroid coordinates by maximizing the likelihood assuming the computed membership values:

$$C_k = \frac{\sum_{n=1}^{N} u_k(\mathbf{x}_n)^m \mathbf{x}_n}{\sum_{n=1}^{N} u_k(\mathbf{x}_n)^m}$$

These steps are then repeated until convergence, as usual in algorithms based on the EM method. The stopping criteria for the fuzzy c-means algorithm are generally either reaching a predetermined number of iterations or the change in the centroid positions falling below some initially specified value.

The result is similar to a k-means clustering, but with continuous membership values. Figure 20.2 illustrates the clustering along with the plot of the membership function for each cluster.

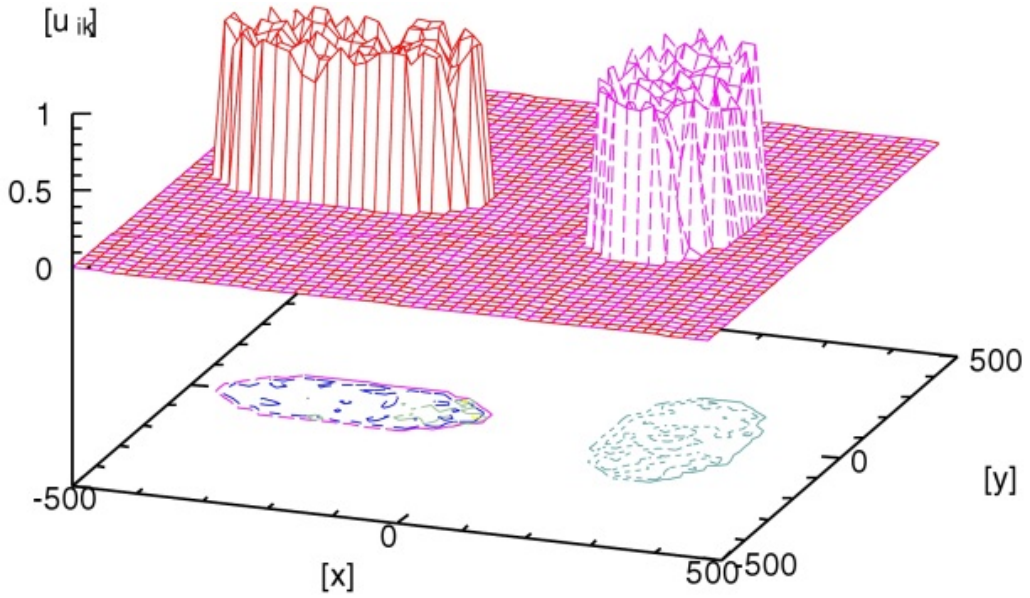

Figure 20.2: Fuzzy c-means example, from "Simulated Annealing - Advances, Applications and Hybridizations", Ed. Marcos de Sales Guerra Tsuzuki, CC BY 3.0

To convert a fuzzy clustering into a crisp clustering — *defuzzification* — it is simply necessary to convert the continuous membership function into $\{0, 1\}$ crisp membership values. This can be done by, for each data point, setting to 1 the largest membership value, and to 0 the remainder, thus assigning the data point to the cluster to which it has the largest membership, or assigning the data point to the cluster with the nearest centroid, as in the k-means algorithm.

20.2 Probabilistic Clustering

Suppose we have a probabilistic model for the distribution of our data, X, given some random distributions from which our data points are drawn, Y:

$$P(X, Y) = P(X|Y)P(Y)$$

This model would allow us to cluster examples from X from the probability of each point given each of the Y distributions and, from this joint probability, we would also be able to generate new points. The problem is to find Y.

Let us suppose that this abstract Y can be decomposed into a set of parameters θ and a set of latent (hidden) variables Z. The likelihood function for our parameters θ given complete information on X and Z would be:

$$L(\theta; X, Z) = p(X, Z|\theta)$$

We do not know Z, but we can use the Expectation-Maximization method to solve this problem iteratively, as we saw before.

Let us assume that our data set X was drawn from a set of Gaussian distributions. In one dimension, the Gaussian distribution is characterized by the means, μ, and standard deviation σ:

$$\mathcal{N}(x|\mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

We can create a mixture of Gaussian distributions by adding different Gaussian distributions with different weights, so that the weights all add up to 1. In more than one dimension, the Gaussian distributions are:

$$\mathcal{N}(x|\mu, \Sigma) = \frac{1}{(2\pi|\Sigma|)^{1/2}} e^{-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)}$$

Where Σ is now the covariance matrix. A mixture model of k gaussians is:

$$p(x) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x|\mu_k, \Sigma_k)$$

</p> where the π_k are the mixing coefficients, or the weight of each Gaussian distribution, and they have to be such that their sum is equal to one, so that the probabilities are normalized:

$$\sum_{k=1}^{K} \pi_k = 1 \qquad 0 \leq \pi_k \leq 1 \; \forall k$$

For each data point $\mathbf{x}$, let us create a variable $\mathbf{z}$ so that:

$$z_k \in \{0, 1\} \qquad \sum_k z_k = 1$$

In other words, $\mathbf{z}$ has a value of 1 in one of its dimensions and 0 in all other, with the dimensions corresponding to the Gaussian distributions. This means that $\mathbf{z}$ specifies from which Gaussian distribution the point $\mathbf{x}$ was drawn. We can also define the joint distribution of $\mathbf{x}$ and $\mathbf{z}$ to be $p(\mathbf{x}, \mathbf{z}) = p(\mathbf{z})p(\mathbf{x}|\mathbf{z})$. The marginal probability of any z_k being equal to one, marginalized for all possible values of $\mathbf{x}$, is equal to the weight of the corresponding Gaussian:

$$p(z_k = 1) = \pi_k$$

Since $z_k = 1$ for a particular point means that point was drawn from distribution k, then the conditional probability of the points given $z_k = 1$ is the respective Gaussian:

$$p(\mathbf{x}|z_k = 1) = \mathcal{N}(\mathbf{x}|\mu_k, \Sigma_k)$$

This checks with our initial assumption regarding the probability of $\mathbf{x}$ as drawn from a mixture of gaussians:

$$p(\mathbf{x}) = \sum_z p(\mathbf{z})p(\mathbf{x}|\mathbf{z}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}|\mu_k, \Sigma_k)$$

but now we can explicitly include the latent variables $\mathbf{z}$ and thus solve this problem with the EM method. Using Bayes' rule, we know that the probability of $z_k = 1$ given the point $\mathbf{x}$ is:

$$p(z_k = 1|\mathbf{x}) = \frac{p(z_k = 1)p(\mathbf{x}|z_k = 1)}{\sum_{j=1}^{K} p(z_j = 1)p(\mathbf{x}|z_j = 1)}$$

where the denominator is the marginal probability of the point $\mathbf{x}$, marginalized over all possible source distributions for that point. Given what we know about the probability distribution of $\mathbf{x}$, we can rewrite this as:

$$p(z_k = 1|\mathbf{x}) = \frac{\pi_k \mathcal{N}(\mathbf{x}|\mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}|\mu_j, \Sigma_j)}$$

This is the posterior probability that Gaussian k is responsible for point $\mathbf{x}$:

$$\gamma(z_{nk}) = \frac{\pi_k \mathcal{N}(\mathbf{x}_n|\mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}_n|\mu_j, \Sigma_j)} \tag{20.1}$$

The log-likelihood of our parameters π, μ and Σ, describing, respectively, the weights, means and covariance matrices of our Gaussian distributions in the mixture, is the log-probability of the observed data given the parameters:

$$\ln p(\mathbf{X}|\pi, \mu, \mathbf{\Sigma}) = \sum_{n=1}^{N} \ln \left(\sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_n|\mu_k, \Sigma_k) \right)$$

The derivative of the log-likelihood function is zero at:

$$0 = -\sum_{n=1}^{N} \frac{\pi_k \mathcal{N}(\mathbf{x}|\mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}|\mu_j, \Sigma_j)} \sum_k (\mathbf{x}_n - \mu_k)$$

Which we can rewrite as a function of μ_k using Equation 20.1:

$$\mu_k = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk})\mathbf{x}_n$$

Where N_k is the effective number of points in component k:

$$N_k = \sum_{n=1}^{N} \gamma(z_{nk})$$

<p>Doing the same for Σ_k, we get this expression:

$$\Sigma_k = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk})(\mathbf{x}_n - \mu_k)(\mathbf{x}_n - \mu_k)^T$$

which is the sum of each covariance weighted by the corresponding $\gamma(z_{nk})$. Finally, for the parameters π_k:

$$\pi_k = \frac{N_k}{N}$$

This means that we can find the maximum likelihood solution of our parameters π, μ and Σ if we know the values of the $\gamma(z_{nk})$, which in turn depend both on X and our parameters. Once again, the solution is to use the Expectation-Maximization method. Starting from random values for π, μ and Σ, we iteratively recompute the $\gamma(z_{nk})$ and recompute the parameters until convergence. Figure 20.3 shows what happens to the Gaussian distributions after one, two and three passes of this method.

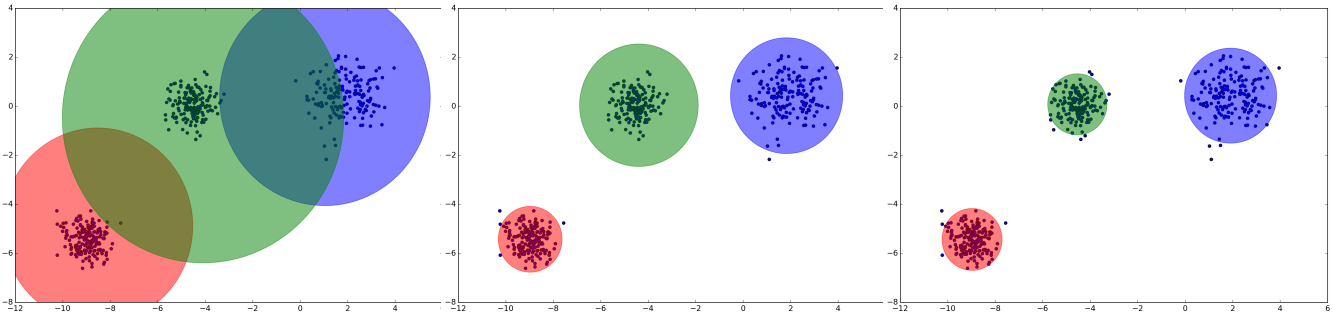

Figure 20.3: Three iterations of the EM method for the computation of a mixture model of 3 Gaussian distributions.

20.3 The Rand index

Previously, we saw the silhouette score as an internal index to evaluate clusterings, and we also talked about the possibility of using external indexes. The Rand index is an example of an external index we can use when we want to compare a clustering with some other partition of our data, such as another clustering, classification labels or any other way of organizing our data into different groups.

Let us suppose our N examples are grouped into some partition X composed of groups $\{X_1, X_2, X_3, ...\}$ and we have a clustering Y with clusters $\{Y_1, Y_2, Y_3, ...\}$. Note that this is not a supervised learning problem, so we are not trying to predict the exact groups each example will fall into. However, we would like our clustering to place in the same cluster of Y examples that belong in the same group of X and in different clusters points that belong to different groups.

To measure this, we can consider all $N \times (N-1)/2$ pairs of examples and label any pair "positive" if the two examples belong in the same group of X and "negative" if they belong to different groups. This way, we can make an analogy to the true and false positives, and true and false negatives, of supervised learning:

- True Positive: a pair of examples from the same group placed in the same cluster
- True Negative: a pair of examples from different groups placed in different clusters
- False Positive: a pair of examples from different groups placed in the same cluster

- False Negative: a pair of examples from the same group placed in different clusters

This makes it easy to understand the Rand index as analogous to the accuracy of a classifier:

$$Rand = \frac{TP + TN}{TP + TN + FP + FN} = \frac{TP + TN}{N(N-1)/2}$$

One shortcoming of the Rand index is that it does not account for the possibility of the clustering of pairs of examples matching the groups with which we compare them. The Adjusted Rand Index solves this problem by subtracting the expected index values if the clustering was uncorrelated to the groups it is being compared to. The Adjusted Rand Index varies from -1 to 1, with 0 indicating no correlation, and can be computed in Scikit Learn using the `adjusted_rand_score` function from the `sklearn.metrics` module.

Following this analogy with classification, we can also compute scores analogous to precision, recall and the F1 measure:

$$Precision = \frac{TP}{FP + TP} \qquad Recall = \frac{TP}{FN + TP} \qquad F1 = 2\frac{Precision \times Recall}{Precision + Recall}$$

20.4 Further Reading

1. Bishop [4], Section 9.2
2. Scikit-Learn demo on Mixture Models: http://scikit-learn.org/stable/modules/mixture.html

Bibliography

[1] Uri Alon, Naama Barkai, Daniel A Notterman, Kurt Gish, Suzanne Ybarra, Daniel Mack, and Arnold J Levine. Broad patterns of gene expression revealed by clustering analysis of tumor and normal colon tissues probed by oligonucleotide arrays. *Proceedings of the National Academy of Sciences*, 96(12):6745–6750, 1999.

[2] Ethem Alpaydin. *Introduction to Machine Learning*. The MIT Press, 2nd edition, 2010.

[3] David F Andrews. Plots of high-dimensional data. *Biometrics*, pages 125–136, 1972.

[4] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer, New York, 1st ed. edition, oct 2006.

[5] Deng Cai, Xiaofei He, Zhiwei Li, Wei-Ying Ma, and Ji-Rong Wen. Hierarchical clustering of www image search results using visual. Association for Computing Machinery, Inc., October 2004.

[6] Guanghua Chi, Yu Liu, and Haishandbscan Wu. Ghost cities analysis based on positioning data in china. *arXiv preprint arXiv:1510.08505*, 2015.

[7] Le Cun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Handwritten digit recognition with a back-propagation network. In *Advances in Neural Information Processing Systems*, pages 396–404. Morgan Kaufmann, 1990.

[8] Pedro Domingos. A unified bias-variance decomposition. In *Proceedings of 17th International Conference on Machine Learning. Stanford CA Morgan Kaufmann*, pages 231–238, 2000.

[9] Hakan Erdogan, Ruhi Sarikaya, Stanley F Chen, Yuqing Gao, and Michael Picheny. Using semantic analysis to improve speech recognition performance. *Computer Speech & Language*, 19(3):321–343, 2005.

[10] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996.

[11] Brendan J Frey and Delbert Dueck. Clustering by passing messages between data points. *science*, 315(5814):972–976, 2007.

[12] Arthur E Hoerl and Robert W Kennard. Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics*, 12(1):55–67, 1970.

[13] Patrick Hoffman, Georges Grinstein, Kenneth Marx, Ivo Grosse, and Eugene Stanley. Dna visual and analytic data mining. In *Visualization'97., Proceedings*, pages 437–441. IEEE, 1997.

[14] Chang-Hwan Lee, Fernando Gutierrez, and Dejing Dou. Calculating feature weights in naive bayes with kullback-leibler measure. In *Data Mining (ICDM), 2011 IEEE 11th International Conference on*, pages 1146–1151. IEEE, 2011.

[15] Stuart Lloyd. Least squares quantization in pcm. *Information Theory, IEEE Transactions on*, 28(2):129–137, 1982.

[16] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA., 1967.

[17] Stephen Marsland. *Machine Learning: An Algorithmic Perspective*. Chapman & Hall/CRC, 1st edition, 2009.

[18] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.

[19] Yvan Saeys, Iñaki Inza, and Pedro Larrañaga. A review of feature selection techniques in bioinformatics. *bioinformatics*, 23(19):2507–2517, 2007.

[20] Roberto Valenti, Nicu Sebe, Theo Gevers, and Ira Cohen. Machine learning techniques for face analysis. In Matthieu Cord and Pádraig Cunningham, editors, *Machine Learning Techniques for Multimedia*, Cognitive Technologies, pages 159–187. Springer Berlin Heidelberg, 2008.

[21] Giorgio Valentini and Thomas G Dietterich. Bias-variance analysis of support vector machines for the development of svm-based ensemble methods. *The Journal of Machine Learning Research*, 5:725–775, 2004.

[22] Jake VanderPlas. Frequentism and bayesianism: a python-driven primer. *arXiv preprint arXiv:1411.5018*, 2014.